UNIVERSITY OF ALBERTA

A Framework for Regression Testing

by

Hareton Kam Nang Leung ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

SPRING, 1992

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-73093-5

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Hareton Kam Nang Leung
TITLE OF THESIS: A Framework for Regression Testing
DEGREE: Doctor of Philosophy
YEAR THIS DEGREE GRANTED: 1992

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Hareton Leung

5638 Lanark Street
Vancouver, British Columbia
Canada V5P 2Y3

March 18, 1992

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A Framework for Regression Testing** submitted by Hareton Kam Nang Leung in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dr. Piotr Rudnicki

Lee White
Dr. Lee White

Stan Cabay
Dr. Stan Cabay

J Schaeffer
Dr. Jonathan Schaeffer

John H. Rowland
Dr. John Rowland

Jack Mowchenko
Dr. Jack Mowchenko

March 18, 1992

ABSTRACT

Regression testing is a testing process used to revalidate a modified software system. Current research has focused mainly on unit (module) level regression testing. This dissertation develops a comprehensive strategy for software regression testing at unit, integration and system level, based on selecting a subset of the previous tests (*selective retest*). As part of the development of these strategies, I have also classified the common errors and faults in combining modules into a working system, and describe a test selection strategy for integration. The concept of a *firewall* is introduced to reduce regression integration effort. It is shown that re-integration of all modules is not needed for all modifications. This result implies that large saving can be realized through selective retest.

The proposed regression testing framework has been evaluated based on four successive modifications of a program. The experimentation shows that this strategy is more cost-effective than the *retest-all* strategy which reruns all previous test cases, using only 34 percent of the tests used by the retest-all strategy.

Global variables continue to be widely used despite their many undesirable effects on software maintenance. This dissertation also analyzes the complication in regression testing associated with the use of global variables. It is shown that global variables testing is not more difficult than parameters testing.

This dissertation also answers the question on the economical advantage of using a selective retest strategy and the retest-all strategy. A cost model has been developed and the conditions under which the selective strategy will be more economical than the retest-all strategy have been established. This result can be used to identify the most cost-effective regression strategy to be used for a given change situation.

Acknowledgements

I would like to express my deepest appreciation to my thesis supervisors: Dr. Lee White and Dr. Piotr Rudnicki for their guidance, encouragement and continuous support.

I would like to thank other members of the supervisory committee: Dr. Stan Cabay and Dr. Jonathan Schaeffer for their critical review and suggestions which helped to improve the thesis.

I would like to thank Dr. Jack Mowchenko, Dr. Paul Sorenson and Michael Lewchuk for many constructive comments, and Paul Godreau, Allan Randall, and Dr. Keith Smillie for supplying the test programs.

Table of Contents

Page

Chapter One: Introduction 1

1.1. Research Overview 2

1.2. Basic Concepts and Terminology 4

1.3. Regression Testing 6

1.4. Relationship to Testing 7

1.4.1. Testing Techniques 7

1.4.2. Differences between Testing and Regression Testing 9

1.4.3. Similarities between Testing and Regression Testing 10

1.5. Review of Regression Testing Methods 10

1.6. Overview of Dissertation 12

Chapter Two: Characterization of Regression Testing 14

2.1. Regression Testing Assumptions 14

2.2. Types of Regression Testing 19

2.3. Principles of Regression Testing 21

2.4. Problem Decomposition 22

Chapter Three: Impacts of Modifications on the Test Set 24

3.1. Test Classes for Regression Testing 25

3.1.1. Evolution of a Test Set 26

3.1.2. Levels of Use of Tests 27

3.1.3. An Example 29

3.2. Non-redundant Test Set 30

Chapter Four: Regression Testing at the Unit Level 35

4.1. Corrective Regression Unit Testing Strategy (CRuT) 35

4.2. Progressive Regression Unit Testing Strategy (PRuT) 38

4.2.1. A Strategy for Progressive Regression Unit Testing 40

4.2.2. Test Selection Considerations 41

Chapter Five: Regression Testing at the Integration Level 44

5.1. An Integration Testing Model 45
5.2. Common Errors and Faults in Calling Another Module 47
5.3. Selecting Tests for Integration Testing 49
    5.3.1. Interface Tests 50
    5.3.2. Functional Tests 50
5.4. Regression Integration Testing 51
    5.4.1. Independent Instructions 52
    5.4.2. Basis Cases for Re-integration of Two Modules 53
5.5. The Notion of a Firewall 59
    5.5.1. Constructing a Firewall 63
    5.5.2. An Example 65
    5.5.3. Practical Application of the Firewall Concept 65
Chapter Six: Regression Testing at the System Level 67
6.1. A Model for System Testing 67
6.2. A Test Coverage Criterion 68
    6.2.1. Interface Subcriterion 70
    6.2.2. Structural Subcriterion 73
6.3. Regression System Testing 79
    6.3.1. Corrective Regression System Testing (CRsT) 80
    6.3.2. Progressive Regression System Testing (PRsT) 80
Chapter Seven: Regression Testing Programs Containing Global Variables 84
7.1. A Testing Model 85
7.2. Basis Cases for Testing Parameters 87
7.3. Global Variables as Parameters 88
7.4. Testing Global Variables 91
7.5. A Regression Testing Strategy for Global Variables 92
Chapter Eight: A Regression Testing System and a Cost Model 96
8.1. An Overview of ReTestS 96
8.2. Components of ReTestS 99

8.3. Comparing the Relative Cost of Regression Testing Strategies............ 101

    8.3.1. A Test Cost Model 102

    8.3.2. Cost of Regression Strategies.. 103

Chapter Nine: Empirical Results.. 107

9.1. Applying the Regression Strategy.. 107

    9.1.1. The Program Under Test... 107

    9.1.2. The Experiment.. 108

    9.1.3. Testing the Original Program 110

    9.1.4. Program Modifications and Regression Tests 111

    9.1.5. Results.. 114

        9.1.5.1. Test Effort Results.. 115

        9.1.5.2. Test Effectiveness Results..................................... 115

        9.1.5.3. Additional Results 116

9.2. Test Cost Study.. 117

    9.2.1. Test Selection and Result Checking Costs 118

    9.2.2. Test Execution Cost.. 119

Chapter Ten: Concluding Remarks.. 120

10.1. Summary 120

10.2. Future Research 122

References 124

Appendix I: An Overview of Integration Strategies.............................. 130

Appendix II: Independence of Regression Testing and Integration Strategies 135

Appendix III: Glossary of Terms and Symbols 137

VITA 141

List of Tables

Page

1. Differences between Corrective and Progressive Regression Testing 21
2. Classification of Tests According to Changes 26
3. The Effect of Modification on the Test Plan 40
4. Breakdown of Tests Used for Regression Integration Testing 58
5. Representing a Global Variable as a Parameter 91
6. Modification Characteristics and Regression Tests 114
7. Detected Seeded Errors 116
8. Average Execution Time per Test 119

I.1. Integration Order for Bottom-up Testing 131

I.2. Integration Order for Top-down Testing 132

I.3. Integration Order for Sandwich Testing 133

I.4. Integration Order for Build Testing 134

List of Figures

Page

1. Specification and Implementation of FindStudentbyName 20
2. Evolution of a Test Set 28
3. An Example Program 29
4. NTS: Algorithm to Compute a Non-redundant Test Set 32
5. A Component-Test Matrix 33
6. The Regression Testing Process of CRuT 36
7. Notations for Progressive Regression Testing 39
8. Algorithm for PRuT 42
9. Algorithm for Computing Scope of Influence 53
10. Boundary Cases for Firewall Construction 60
11. A Use of Basis Cases and a Calculation of a Firewall 66
12. Data Structures Used for System Testing 69
13. A Subsection of the Input Graph of StudentDatabase 72
14. Constructing a Calling Order Graph 75
15. The Procedure for Constructing a Calling Order Graph 76
16. An Example Construction of a Calling Order Graph 76
17. Calling Order Graph of Program StudentDatabase 78
18. The Procedure for Corrective Regression System Testing 81
19. The Procedure for Progressive Regression System Testing 83
20. Potential Effect of a Global Variable on Other Modules 84
21. Different Pairs of References to a Global Variable 89
22. Relations between a Global Variable and Changed Code 93
23. Some Examples Where Global Variable Testing are Required 95
24. Structure of ReTestS 98
25. Cost Relationship Between the Retest-all and Selective Strategies 105
26. Call Graph of Program StudentDatabase 109

I.1. A Section of the Call Graph of Program StudentDatabase 131

Chapter One

Introduction

Many people have realized that software and its applications will evolve as it is adapted to changing environments, changing needs, new concepts and new technologies. Software will grow in the number of functions, components and interfaces. Old modules may be expanded for uses beyond their original design. Thus, software modification and evolution are inevitable [30, 46, 71, 80]. Much of the time spent on software maintenance is in modifying and retesting the software. Efficient and effective regression testing can reduce the cost of maintenance.

Regression testing is a testing process which is applied after a program modification to detect errors introduced during modification and to increase one's confidence that the modified program still meets its specification. It involves exercising the modified program with some existing tests and new tests to re-establish confidence that the program will perform according to the (possibly modified) specifications.

The problem of regression testing is not the same as the problem of testing, as most people have assumed. Regression testing generally involves testing the portion of the program affected by the modification, instead of the whole program. It is needed throughout the life of a product, once after every modification, while testing is done during the development of the product or its enhancements.

The traditional regression testing strategy was to repeat all the previous tests and re-testing all the features of the program [9, 69], even for small modifications. For programming-in-the-large, the cost of retesting the entire system is expensive if attempted after every modification. This practice is becoming increasingly difficult because of the demand for testing the new functionalities and correcting errors with limited computer and human resources.

An efficient solution to regression testing can bring great benefits to software developers. Several studies have indicated that software maintenance and modification can require over 50 percent of the total life-cycle cost [51, 82]. For some projects, the maintenance cost can be higher. For example, for a given weapons system, about 25 percent of the software life-cycle costs are for development, and 75 percent for maintenance [70]. Also, the increasing number of software applications being deployed may turn software maintenance into a major subindustry by the close of the century, if not sooner [15]. Much of this maintenance activity involves modifications to, and regression testing of, existing programs. Therefore the importance of the regression testing process in software development cannot be overstated. Yet, there has been little work done in this significant area.

A good solution to the regression testing problem brings two key benefits:

(1) *Improved product quality*

Given that there is no accepted strategy for regression testing and the practitioners seem to retest in an *ad hoc* manner, any systematic strategy would, on average, increase the software reliability. The resulting product should have fewer errors and fewer failures. Another benefit of following a systematic strategy is that the software can be subjected to the same degree of retesting by anyone who follows the strategy. The process of retesting no longer depends on the experience and expertise of test analysts. As a result, software developed with the strategy will have the same predictable quality.

(2) *Reduced maintenance cost*

Regression testing is an essential component of every maintenance job. If we can find a method which requires less testing, but also thoroughly tests the program, the saving in human and machine resources can be tremendous.

1.1. Research Overview

Although regression testing is a key component of software development and maintenance, the state of the practice is highly individualized and no systematic strategy is followed. Regression testing can be made more efficient with strategies that emphasize partial retesting rather than rerunning all existing tests after a software modification (that is, the *retest-all* strategy). The retest-all strategy is usually costly due to program size and testing frequency, but this is currently a common practice. The underlying hypothesis of partial retesting is that a small modification to a system often has a small impact and therefore a "focused" retest will be more efficient than the retest-all strategy and equally effective.

This dissertation represents a fundamental study of the regression testing problem. An effective, efficient and systematic approach to regression testing has been developed. In particular, I have developed a regression testing framework which can provide a significant saving over the retest-all strategy. Although regression testing is currently receiving more attention, most proposed methods are restricted to testing at the unit level [3, 23, 79]. There is a lack of research in strategies for regression integration and regression system testing. In this dissertation, I have analyzed all three phases of the regression testing problem: unit, integration and system testing, and developed regression testing strategies for each of these phases.

Two types of regression testing have been identified: one involves only implementation changes while the other includes specification changes. This classification is important because each type of regression testing requires different degree of testing effort. The first type of regression testing requires a simpler testing process than the second type because more tests can be reused.

The effect of system modifications on the existing test set is also studied. The tests can be categorized into five major classes depending on the modification. This classification provides a basis for the test selection process. By focusing on the impact of the modifications on the existing tests, it is possible to identify which tests will be useful in retesting the system, which tests have become obsolete, and which tests should not be repeated because they will give the same results as before.

Integration testing is an important phase of the testing process which has not been thoroughly studied. Although integration strategies are well-understood, there is a lack of research on the test selection problem. This dissertation has developed an error model and identified the common errors in software integration. These errors usually arise because of a misunderstanding of the specification of the called module. I have developed a regression testing strategy based on the integration testing of a set of basis cases involving a pair of modules with at least one of them modified. Various use levels of the integration tests have also been identified. These results can be used to estimate the retesting effort for a given change situation.

The notion of a *firewall* is introduced to encapsulate all the modules which should be re-integrated after a modification. It is shown that a complete re-integration of all modules is generally not required. This result can be used to substantially reduce the retesting effort while maintaining the same test effectiveness.

Before developing a regression system testing strategy, I first introduce a test coverage criterion which can be used as a termination criterion for system testing. All too often, system testing is based purely on functional tests. My system testing strategy is based on both functional testing and the testing of the user interfaces and a special set of module combinations derived from the calling order of modules. An experimentation of the retesting framework indicates that this system testing strategy is easy to apply and effective in detecting faults.

Despite the existence of global variables in many software systems, testing global variables has traditionally been overlooked by the testing community. A *global variable* is a variable which is referenced by a module other than the one containing its declaration. This dissertation also analyzes the problem of regression testing programs that contain global variables. It is shown that global variables can be treated as extra parameters for testing purposes. Thus, global variable testing should not be more difficult than parameter testing.

The dissertation also answers the question of the relative cost benefits of the selective retest and retest-all strategies. A cost model for comparing regression testing strategies has been proposed. This model is based on realistic assumptions that are supported by empirical data. The conditions under which a selective retest strategy is more economical than the retest-all strategy has been established. The result can be used to identify the most cost-effective retesting strategy to be used for a given change situation.

Finally, the proposed framework was used to test four successive modifications of a 550 line Pascal program and the results indicated that this framework can save over 65 percent of the tests required by the retest-all strategy, while providing the same test effectiveness. This experiment suggested that the strategy is superior to the retest-all strategy since it requires fewer tests and has the same test effectiveness. Empirical data was also collected to validate the cost model assumptions. The data from a case study suggested that the assumptions on the dependency of selection, execution and result checking costs on the number of tests are valid.

The remainder of this chapter describes the background information for the research. Section 1.2 introduces the basic concepts and terminology to be used throughout the dissertation. Section 1.3 describes regression testing. Section 1.4 gives an overview of the testing techniques and discusses the relationship between regression testing and testing. Section 1.5 reviews several regression testing methods and points out their common weaknesses. Since many new concepts and terms will be introduced in the following sections and chapters, a glossary is provided in Appendix III for easy reference.

1.2. Basic Concepts and Terminology

To facilitate the presentation, this section defines the basic concepts and terminology that will be used. A *module* is a discrete and identifiable part of a program with respect to compiling, combining and loading with other program units. A module may invoke other modules and may be invoked by other modules. A module can be viewed as a logically separable part of a program and is constructed to encapsulate a single system function or operation such as sorting a list of names in alphabetical order. Our view of module is more limited than other definitions. For our purposes, a module is not the same as a *class* in SIMULA, a *module* in MODULA, or a *package* in Ada, but is like a *procedure* or a *function* in Pascal.

A *call instruction* is an instruction which invokes another module; the common syntax of a call instruction is the name of the called module, followed by the actual parameter list. For simplicity of presentation, the module parameters will not be shown unless they are required for the discussion. *Call(B)* will be used to denote a call instruction to module B.

A *modified* module is a module which has undergone some modifications such as additions, deletions, or changes to some instructions within the module. A *new* module is a module which was added to the program. A *deleted* module is an existing module which was removed from the program. An *affected* module refers to a modified, new or deleted module. Similarly, an *affected* instruction refers to a modified, new or deleted instruction.

A program usually consists of a number of modules. The calling relationship between these modules can be represented by a *call graph* or a *structure chart*. A *call graph* is a directed graph showing the control hierarchy of a program with rectangles representing modules. An arrow from module A to module B indicates that module A may call module B. Module A is an *ancestor* of module B if there exists a path (a sequence of calls) in the call graph from module A to B; B will be called a *descendant* of A.

A *structure chart* describes the system as a hierarchy of parts in the form of a tree. It is derived from a data-flow diagram with each component formed from those components lower in the tree. Each component may become a program module and is represented by a rectangle. The links between rectangles are labeled with annotated arrows. An arrow entering a rectangle implies input and that leaving a rectangle implies output. For the following analysis, a call graph is sufficient to express the calling relationships between modules.

To simplify the analysis, we assume that there is no recursive calls in the program and therefore the call graph will be a tree. Observe that the call graph does not show the invocation order of the modules during an execution of the program. A new graph structure that captures this information will be introduced in Chapter Six.

The control structure of a module can be represented by a *control flow graph*, $G = (N, E, n_s, n_f)$, where N is a set of nodes and E is a set of edges in $N \times N$. Each node in the graph represents an executable *instruction*, while the edge, denoted by (n_i, n_j), indicates that a possible transfer of control exists from node n_i to node n_j. A node of the control flow graph can represent an assignment statement, an input or output statement, the conditional expression of an **if-then-else**, or the conditional expression of a **while** statement. For example, a node representing a conditional expression will be called a *conditional node*. The conditional expression will be treated as an instruction, called a *conditional instruction*.

We assume there exists a single entry point, the start node, n_s, and a single exit point, the final node, n_f, in the control flow graph. If necessary a null node can be added to the graph for the start node and likewise for the final node to convert the graph into single-entry, single-exit.

A *subpath* from n_i to n_{i+k-1} of length k is a list of nodes $(n_i, ..., n_{i+k-1})$ such that for all j, $i \leq j \leq i+k-2$, $(n_j, n_{j+1}) \in E$ and not all n_j are necessarily distinct. A *path* is a subpath that begins at the start node, n_s, and ends at the final node, n_f. Some paths may be nonexecutable due to contradictory conditions on the transfer of control along them. A path is *feasible* if there exists input data which causes the path to be traversed during program execution. Each module is composed of a set of paths. Some modules may have an infinite number of paths due to the presence of program loops. The control flow graph represents all possible paths in a module.

An *instruction trajectory* is a feasible path that can actually be executed for some input. An instruction trajectory can be used to compute the structural coverage of a test set. We will consider an instruction trajectory modified if any instruction in the trajectory is modified even though it contains the same sequence of instruction identifiers. A *module trajectory* is a trace of modules executed by a single test. A module trajectory is similar to an instruction trajectory except that the elements recorded are module identifiers rather than instruction identifiers.

A *use* of variable v is an instruction in which this variable is referenced. A *definition* of variable v is an instruction which assigns a value to v. Let USE(J) be a set of variables whose values are used in instruction J and DEF(J) be a set of variables whose values are defined in instruction J [28]. A path $(n_x, n_i, ..., n_j, n_y)$ containing no definition of variable v in nodes $n_i, ..., n_j$ is called a *definition-clear path* with respect to (w.r.t.) v from node n_x to node n_y.

Given instructions J and K, there is a *direct definition-use relation* from J to K (written as J>>K) if there exists a variable v such that

(1) $v \in$ USE(K),
(2) $v \in$ DEF(J), and
(3) there exists a definition-clear path from n_J to n_K w.r.t. v.

Given instructions J and K, there is an *indirect definition-use relation* from J to K if there exists instructions $L_1, ..., L_n$, $n \geq 1$, such that J >> L_1, L_{i-1} >> L_i, L_n >> K, $1 < i \leq n$.

A *program component* is any subset of instructions of a program. It can be a single instruction, a branch, a *segment* (basic block), a sequence of instructions between a definition and a use of the same variable, or two instructions with one definition and another use of the same variable. This notion of a program component is more general than that of Weyuker [73]. Weyuker's component of a program is a contiguous sequence of statements. By her definition, a component is a section of the program which is single-entrant, and may consist of several subpaths. Weyuker's program component is based on the program text, while ours is based on the subpaths of the control flow graph and does not need to be a contiguous sequence of statements.

The terms *error, fault,* and *failure* commonly cause confusion. An *error* is a mental mistake made by a programmer or designer. A *fault* is a software defect which can cause a failure. A *failure* occurs whenever the software system fails to perform the required function according to its specification.

1.3. Regression Testing

Regression testing is a major component in the maintenance phase where the software system may be corrected, adapted to new environments, or fine-tuned and enhanced to improve its performance. Software maintenance is defined as the activities required to keep a software system operational and responsive after it is accepted and placed into production [52].

There are three types of modifications, each arising from a different type of maintenance. According to Lientz and Swanson [50], *corrective maintenance*, commonly called a "fix", involves correcting software failures, performance failures, and implementation failures to keep the system working properly. Adapting the system in response to changing data requirements or processing environments constitutes *adaptive maintenance*. Finally, *perfective maintenance* covers any enhancements to improve the system processing efficiency or maintainability. A study by Glass [20] has shown that 60 percent of the maintenance effort is spent improving the software product, 18 percent is spent on changes required by environmental factors, 17 percent is spent fixing errors, and 5 percent is spent elsewhere.

Regression testing is also needed during the late stages of development, since changes to a software system are not confined to the maintenance phase. Many changes are made to the software system prior to delivery. After the coding phase, and near the end of the testing phase, any change may entail regression testing, rather than testing anew. At this stage, a well-developed test plan should be available. It makes sense to reuse the existing tests, rather than designing all new tests, in retesting the program after it is corrected for any errors.

Modification may involve implementation change, specification change, or both. The code change may involve minor modifications such as adding, deleting, rewriting a few lines of code, or major modifications such as adding, deleting or replacing one or more modules or subsystems. During adaptive or perfective maintenance, new modules are usually introduced. The specification of the system should be modified to reflect the required improvement or adaptation. However, in corrective maintenance, the specification is not likely to be changed and no new modules are likely to be introduced. Most modifications involve adding, deleting and modifying instructions. Many program modifications occurring during the development phase are similar to that of corrective maintenance when faults identified during code inspection and testing are corrected. The specification is unlikely to be modified for this kind of code modification. However, if a conceptual design error was made or the specification was misunderstood, the resulting program modification is similar to the case of changing the program specification. In this case, the program modification may be viewed as adaptive or perfective maintenance.

1.4. Relationship to Testing

Many people have assumed that regression testing is just a simple extension of testing and there are no major differences between these two processes. This section first reviews two common testing techniques and then compares the regression testing process to that of testing. It will be shown that although regression testing is different from testing in many respects, they both possess the same important characteristics.

1.4.1. Testing Techniques

There are two common testing methods. The first method is to generate tests based on the specification, that is, black-box testing. One common specification-based testing method is *functional testing* as described recently by Howden [34, 35]. The key idea in *functional testing* is that a program is considered as constructed of a series of functions, ranging from very low level to high level functions, which are a synthesis of low level functions. Program functions are represented by program components and represent different programming ideas which should be individually tested. Testing is complete when all elementary statement-level, intermediate design-level, and program-level functions are tested.

Howden described two types of functions: *requirements functions* and *design functions*. Requirements functions describe the overall functional capabilities of a program, while design functions are used to implement requirements functions. For example, a text-string processor requirements function may need design functions such as *get-next-char-type*, *extract-words* and *word-length*.

In functional testing, the test analyst first identifies the functions which are supposed to be implemented by the program, and then selects test data that can be used to check that the program implements the functions correctly. The functions may be identified from the design documents if they are available, or reconstructed from the code and specification documents. Guidelines for the identification of functions and the construction of tests have been developed by Howden [31, 34]. According to these guidelines, it is obvious that both the identification of functions and the selection of tests require human guidance.

The second common testing method is white-box testing, which bases test selection on the source code. The most popular white-box testing is structural testing which selects tests based on the control or data flow structures of a program [36, 44, 45, 59, 61, 62]. Structural testing (for example, statement, branch, or data flow measures such as data contexts [44], or required-pairs [59]) requires the test analyst to design tests to satisfy a testing criterion, which usually requires certain components of a program, or some combinations of them to be exercised. In practice, one test is selected to exercise each component once and rarely more than once. If no error has been detected using this testing strategy, then the test analyst's confidence in the *reliability* of the program is increased. Software *reliability* is the probability that a software system will operate without failure for a specified time in a specified environment. Tests created from black-box testing techniques will be called *specification-based tests* and those from structural testing *structural-based tests*.

Specification-based tests may be used as structural-based tests, but the converse is not true. Since all specification-based tests execute some components of a program, their trajectory results can be included for structural coverage measures. Consequently, specification-based tests may be used to satisfy the structural coverage criteria. However, not all structural-based tests can be used as specification-based tests because some of them are designed to test a certain program component and this program component may have no functional meaning by itself; it is solely used with other components to synthesize a function.

Both structural-based and specification-based testing methods have deficiencies. In structural-based testing, one test is usually selected to exercise each program component. However, certain errors along a path can only be detected if the path is executed with specific values from its input subdomain. Thus executing each program component with *one* test may not be sufficient to discover these errors. Another shortcoming of structural testing strategies is that if a part of the system specifications is not implemented by the program, then it will not be tested. For example, the control structure of the program may be incorrect so that a part of the specification is not implemented. Consequently, structural testing will not have tests for testing this part of the specification.

Specification-based testing strategies also have their own weaknesses. For example, there may be errors in the code that are encountered only under conditions specific to the implementation, but these conditions are not indicated in the specification. Specification-based testing will not create tests to test this section of the code.

Note that exercising the same program component or the same function with multiple tests is necessary in some situations, although this is rarely done in practice. For example, for functions which are critical to the operation of the system, it is advisable to run more tests to validate its correctness. Another situation that would warrant more testing is when a certain program component is expected to be error-prone. Then more tests should be chosen to exercise this component even though the structural coverage will not be increased.

The input partition concept [64] can be used to illustrate that multiple tests for a program component or a function are beneficial. The test selection strategy of structural testing or functional testing may be viewed as a way of partitioning the input domain. For example, when we use a structural criterion such as all branches, we are grouping all input which exercise a program branch into the same group. However, faults are not generally "partitioned" the same way (with one fault per input partition) as assumed by the structural testing criterion. Thus, there will be cases where more than one failure causing input are present in an input partition. It follows that one test from each input partition may not catch all faults. This supports the common understanding that multiple tests for the same program component or function are beneficial.

Empirical research indicates that using functional testing or structural testing alone cannot detect all errors in a program [32]. Many researchers in the field have advocated that both testing techniques be used to complement each other. A common practice in testing is to first generate specification-based tests, and then augment a test set with additional structural-based tests until the structural coverage criterion is satisfied.

In the following analysis, we will assume the following unit testing strategy is used to test each module. This strategy is similar to the testing practice of leading edge software organizations. During unit testing, each module is tested in isolation, independent of other modules. This testing may require the use of drivers and stubs to simulate the behavior of other interacting modules of the module under test. Both functional and structural-based tests are used. Functional tests are chosen from both the system and design specification, and are executed first. Additional structural-based tests are chosen to satisfy the structural coverage criterion. It is assumed that each program component as required by the structural criterion will be exercised at least once and the test analyst will not try to select more tests than required to satisfy the structural criterion.

1.4.2. Differences between Testing and Regression Testing

The major differences between testing and regression testing are described below:

(1) *Availability of test plan*

Testing begins with a specification, an implementation of the specification and a test plan with tests added during the specification, design and coding phases. All these tests are *new* in the sense that they have not been used to exercise the program previously.

Regression testing starts with a possibly modified specification, a modified program and an old test plan which requires updating. All tests in the test plan should have been previously run and can be assumed to be useful in testing the program.

(2) *Scope of test*

The testing process aims to check the correctness of the whole program, including its individual components and the interactions of these components. Regression testing is concerned with checking the correctness of parts of a program. The portion of a program which is not affected by modifications need not be retested.

(3) *Time allocation*

Testing time is normally budgeted before the development of a product. This time can be as high as half the total product completion time. However, regression testing time is not normally included in the total product cost and schedule. Consequently, when regression testing is done, it is nearly always performed in a crisis situation. The test analyst is normally urged to complete retesting as soon as possible and most often is given limited time to retest.

(4) *Development information*

In testing, knowledge about the software development process is readily available and up-to-date. In fact, the testing group and the development group may be the same. Even if an organization has a separate testing group, test analysts can usually query the developers about any uncertainty in the software. But in regression testing, the program information may not be up-to-date. Since regression testing may be done at a different time and place, the original developers may no longer be available. This situation suggests that any relevant development information should be retained and updated continuously if regression testing is to be successful.

(5) *Completion time*

The completion time for regression testing should normally be less than that for testing since only parts of a program are being tested.

(6) *Frequency*

Testing is an activity which occurs frequently during code production. Once the software product is put into operation, testing is completed and any further testing will be considered as regression testing. Typically, regression testing is applied many times throughout the life of a product, once after every modification.

1.4.3. Similarities between Testing and Regression Testing

Two fundamental aspects of regression testing are similar to that of testing.

Purposes

The purposes of testing and regression testing are quite similar. They both aim to:

(1) increase one's confidence in the correctness of a program, and

(2) detect errors in a program.

Some additional goals of regression testing are to:

(3) preserve the quality of the software

The modified software should be at least as reliable as its previous version; this may be achieved in many ways. One possible method is to insist that the new software version achieves the same structural coverage as the old version (for example, if the previous testing included tests that traversed all branches of the program, then the new tests should also traverse all branches of the new software version). Since the previous structural coverage is judged adequate to ensure reliability, it is logical to apply the same criterion to the new software version.

(4) ensure the continued operation of the software

This is an important goal because some users may become dependent on the software product, and software developers have a responsibility to continue to provide the same service to users.

Testing techniques

Since tests in a test plan depend on the chosen testing technique, the testing technique used by both testing and regression testing should be the same if the regression testing process involves the reuse of existing tests.

Another reason for using the same testing technique is that it is easier to evaluate the quality of two software products if they are tested by the same technique. With the current state of the art, it is difficult to compare the relative effectiveness of two different testing methods. For example, a program which has been successfully tested using symbolic execution [4, 8] is not necessarily more "correct" than the same program tested using functional testing. Thus, to quantify the degree of testing of two versions of the same program, it is essential that the same testing techniques be applied to both of them.

1.5. Review of Regression Testing Methods

Only a few regression testing methods have been introduced in the research literature. Recently, regression testing research emphasized retesting only the modified and new features of the program. Harrold and Soffa [23] describe an *incremental* approach to regression testing. This approach assumes the modules are validated entirely using data flow testing [62]. A major feature of this method is that only the parts of the module changed or affected by a change will be reanalyzed and retested. Although this regression testing method concentrates on unit regression testing using a white-box testing technique (structural testing based on data-flow coverage), it has recently been extended to include *interprocedural testing* by Harrold and Soffa [24]. A data flow testing method is combined with incremental data flow analysis to aid in unit and integration regression testing. Harrold and Soffa is developing a prototype tool which implements their strategy [23, 24].

Another regression testing tool that was based on input partition testing strategy was introduced recently by Yau and Kishimoto [79]. Input partition testing strategy [64] divides the input domain of a module into different classes using both the program specification and code, and requires one test be selected from each input class. The objective of regression testing is to execute each new or changed input partition at least once. Symbolic execution is used to identify those input domains which are not modified and to aid in test generation. This regression testing strategy selects a subset of the previous tests and some new tests to exercise the modified code. Similar to most regression tools, this tool concentrates on unit regression testing.

This approach does not work well for programs which are non-numeric because symbolic execution is better suited to numeric programs than non-numeric programs. Also, because symbolic execution has difficulty in evaluating symbolic expressions for programs with logically complex paths, this approach cannot be applied to such programs.

Benedusi, et al. [3] recently describe a regression testing method for unit structural testing based on path testing strategies. After a module is modified, path difference analysis is performed on the two module versions. Different types of paths (*cancelled, new, modified,* and *unmodified*) are then identified and the appropriate old tests are reused and new tests are created. Like most regression tools, this tool only deals with unit regression testing.

Another regression tool has been introduced by Hartmann and Robson [27]. This tool extends Fischer's method [16] for test selection from the previous test set. Fischer uses zero-one integer programming to find a minimum set of tests which can cover all program segments. Four tables are used to record the control flow relation between program segments, reachability between segments, tests which cover the different segments, and variable use and define information within each segment, respectively. A series of inequality constraint expressions is created, one for each program segment, relating those tests which traverse the segment. The objective function relates the cost of running every test to individual tests. In this model, all costs are assumed to be the same and are set to 1. Given the program segments that have been modified, standard linear-programming techniques are used to solve for the objective function which can then be used to derive the minimum number of tests that must be rerun to validate a given modification. Recall that the linear programming problems may terminate with a non-integer solution. To arrive at a zero-one solution, additional constraints and iterations may be needed.

The prototype tool under development by Hartmann and Robson [27] extends Fischer's method to include programs written in the C programming language, programs with several modules, and segments that have multiple uses of variables. Since this tool also uses zero-one linear programming, it suffers from the same set of computational problems associated with other zero-one linear programming problems. Observe that this tool implicitly assumes a white-box testing technique (segment cover) is used and cannot be generalized to include black-box testing.

We can summarize the characteristics of the regression tools described in the literature as follows:

- Most of them apply one testing method (either white-box or black-box testing).

- Most restrict themselves to unit testing and do not address integration or system testing.
- Most of them are either under development or in a primitive prototype form. None of them is in production use, indicating the difficulty in producing such a tool. This can be partly explained by the fact that any regression tool should be coupled with a testing tool because both regression testing and testing should use the same testing technique.

1.6. Overview of Dissertation

In the next chapter, we describe our testing model which assumes that the previous testing had satisfied the testing criterion and a good test plan is available. Two types of regression testing will be identified: corrective regression testing and progressive regression testing. The principles of our regression testing strategies are described in Section 2.3. Section 2.4 presents a decomposition of the regression testing problem into two subproblems: the test selection problem and the test plan update problem. This research focuses primarily on the test selection problem.

Chapter Three deals with the effect of program modification on a test set. We categorize tests into five classes and show the evolution of a typical test set in terms of these five classes. Depending on the program modification, certain properties of a test may be reused. Section 3.1.2 identifies three use levels of a test, based on the extent of usable information associated with the test. A rarely described property of a test set is the redundancy of tests. A *redundant* test is a structural test which exercises the same program components as those exercised by another group of tests in the test set. Section 3.2 defines a *non-redundant test set* to be a test set with no redundant tests and introduces an algorithm for computing such a test set.

Chapters Four, Five and Six develop strategies for unit, integration and system regression testing with no specific testing for global variables. Since global variables require a new set of considerations, they are treated separately in Chapter Seven. The regression testing strategies for corrective and progressive unit testing are presented in Chapter Four. Section 4.1 gives the regression testing strategy for modifications that are restricted to the implementation and Section 4.2 discusses the corresponding strategy for specification changes.

Chapter Five deals with test selection for integration testing and regression integration testing. Section 5.2 identifies the common errors and faults that may occur when one module calls another module. A common mistake occurs whenever the calling module has an incorrect expectation of the called module. Integration testing objectives and test selection strategy are described in Section 5.3. Section 5.4 presents the regression testing strategies for various basic types of modifications involving two modules with at least one of them modified. The level of use of existing tests has been determined for these basic cases. Section 5.5 introduces the concept of a *firewall*, and shows a procedure for its construction.

In Chapter Six, a test coverage criterion that can be used as a termination criterion for system testing is introduced. System testing is completed when all the software functions are tested and the *structural-interface* criterion is satisfied. A regression testing strategy for system testing is also presented. This strategy emphasizes reusing the previous analysis and tests. Testing expenses are therefore reduced because fewer tests and less analysis are needed.

Global variables have been widely used in practice, despite their undesirable impacts on software maintenance. Chapter Seven studies the complication due to the occurrence of global variables. Insights into regression testing programs that contain global variables are presented. It is shown that global variables can be treated as parameters and tested accordingly.

In Chapter Eight, the design of a regression testing system - ReTestS is outlined. Conceptually, ReTestS may be viewed as a single environment which provides many capabilities useful for performing all aspects of the regression testing process. Section 8.3 analyzes the cost of testing and a model of test cost is developed. In Section 8.3.2, a cost comparison of using a selective retest strategy and the retest-all strategy is presented.

Chapter Nine describes the application of our regression strategy to an actual software system. It was found that our strategy was equally effective as the retest-all strategy, but required fewer tests (using approximately 34% of tests). Also, empirical data was collected to support the assumptions used in building the test cost model in Section 8.3. Finally, a summary of the dissertation and work for future research is given in Chapter Ten.

Chapter Two

Characterization of Regression Testing

Every regression testing strategy has its own set of assumptions: some are implicit while others are clearly stated. In this chapter, we present our regression testing model. In Section 2.1, assumptions about the testing process, test plan and change information are described. This research focuses on functionality modifications and does not consider modifications to resource dependencies and data structures. The *all-essential assumption* is introduced to cut down on the amount of analysis and forms the basis of our test selection criteria. A model for system specification is also outlined in this section.

Section 2.2 classifies regression testing into two major categories: progressive regression testing and corrective regression testing. The former involves modification to the specification while the latter does not and only involves implementation changes. The principles of our regression testing strategy are described in Section 2.3. Section 2.4 breaks down the regression testing problem into two subproblems: the test selection and the test plan update problems. Selecting tests for regression testing depends on three different criteria. A given regression strategy may be characterized by its set of selection criteria.

2.1. Regression Testing Assumptions

Many of the following assumptions are based on the common testing practices.

Testing Process

The testing process is assumed to be structured into three phases: unit testing, integration testing, and system testing. Each phase provides its own effectiveness with respect to detecting certain faults. In a cost-effective testing process, each phase should work in concert with the other phases, and should concentrate on finding faults which cannot be easily detected by other phases. At the same time, for faults which can be detected by several phases, resources should not be expended in duplicating the same testing effort.

At the initial stage of our research, we were looking for new methods for regression testing. However, we quickly realized that the testing method used for both the initial testing and regression testing should be the same based on the *identical procedure principle* to be described in Section 2.3. Thus, this thesis focuses primarily on regression testing and does not attempt to introduce any new testing method, except in those situations when no testing method is available. By assuming that the initial testing has *adequately* tested the program, we then analyze the regression testing problem and develop regression strategies based on the initial testing method. We will not concern ourselves with the intricate details of test selection and strategy during the initial testing.

Many test selection strategies have been advanced [8, 34, 44, 61, 64, 77] and the relative effectiveness of these strategies has been compared [13, 22, 33]. No one strategy has proven to be consistently superior than the others. It has been generally accepted that multiple testing methods such as functional and structural testing should be used together to test the software systems.

It will be assumed that the test set includes both types (specification-based and structural-based) of tests so that each module is unit tested with both functional and structural tests. Furthermore, the functional testing is assumed to have sufficient tests to test the identified functions of the modules [34]. For structural testing, a structural coverage measure is used and tests are selected so as to provide the required coverage [7, 44, 62]. The coverage measure can be any control-flow or data-flow criteria such as all-branches and all define-use pairs.

The key requirement of an *adequate* test set is that it has tested all the functions of the software system, exercised all the required program components for satisfying the structural testing criterion and provided confidence to the test analyst that the system is free of error. If such an adequate test set can be regenerated for a changed program version, then regression testing can terminate since this version has achieved the same "degree of testing" as the previous testing. Guidelines for generating an adequate test set can be found in [10, 34, 38, 44, 62].

This test process model is similar to the current practices of many leading-edge software organizations.

Test Plan

As described in Chapter One, a test plan storing the previous tests is needed if the regression testing strategy aims to reuse some of the previous tests. We will assume that the previous test plan is available. This assumption should hold in most large software organizations, although the information stored in their test plans may not be as extensive as our test plan.

There are three major components in a test plan: *test component*, *dynamic component*, and *static component*. The *test component* stores the tests used and their expected output. The *dynamic component* stores the information collected during test execution such as the instruction and module trajectories; it represents the behavior of the program for different inputs. The *static component* of a test plan records some static information about the program such as the control flow information in the form of a control flow graph, and the data flow information in the form of define-use chains.

Some information that can aid the testing process can be extracted from the program code and specification without exercising the program. Almost all the development information (for example, functional analysis and design trade-off decisions) may be used to design tests. Therefore, it is essential that this information forms part of the test plan and is kept in a database for easy lookup. Also, by performing a static analysis of the source code, information such as the location of branches, loops, and define-use chains can be derived and used for computing the structural coverage measure.

One can actually store less information in the test plan. In fact, the test component is the minimum data that should be stored since the other two components can be regenerated when needed. The dynamic component can be generated by rerunning the tests, and the static component can be obtained by repeating a static analysis of the program. However, since this information is available during testing, it should be stored for later use, so that effort for regeneration can be avoided.

Next, we introduce a simple data structure which is useful for storing the dynamic information of the program. A ***component-test matrix*** records the program components such as instructions, branches, segments, and modules exercised by each test. For example, a branch-test matrix stores the program branches traversed by each test, while a module-test matrix records the modules traversed by each test. The component-test matrix is represented by $[A_{ij}]$, $1 \leq i \leq c$, $1 \leq j \leq t$, where c is the total number of components, t the total number of tests, $A_{ij} = 1$ if test j traverses component i, and $A_{ij} = 0$ otherwise.

The actual component-test matrix produced will depend on the testing phase and the type of structural testing. For unit testing, one can use an instruction-test, a branch-test or a segment-test matrix, and for integration and system testing, a module-test matrix may be more space effective. Fischer, et al. [16] have previously called the segment-test matrix the *test case cross reference matrix*.

The Test Plan assumptions can be summarized as follows:

(1) The Test Plan stores all three components and the dynamic component is represented by several component-test matrices.

(2) The Test Plan includes a ***testing guideline*** which is a complete specification of the testing process given the test design strategy, the coverage measure achieved, and the procedure for handling obsolete tests.

Some test analysts may decide to remove the obsolete tests from the test plan, while others may keep them as illegal tests. The advantage of the latter approach is that repeating the obsolete tests provides some confidence that the program does not regress to the previous versions. However, the test plan tends to grow larger and requires increasingly more testing effort.

Changes information

The test analyst will obtain a list of modifications made to a software product from the software maintainer. In particular, the test analyst is given the modified specification, and a list of modified instructions, from which the modified modules can be determined. Without this information, the test analyst would have to retest the whole program and the cost of testing cannot be reduced. This assumption is realistic and represents the current practices of most software organizations. Some organizations may have automated tools to identify the changes, while others may rely on a manual process.

The All-Essential Assumption

Not every instruction traversed during an execution affects the output. On a particular path, there may be some instructions which do not affect any output variables (in terms of control flow and data flow), while the same instructions may affect output variables on another path. More formally, each instruction in a module can be viewed as performing a subcomputation. Each path can be viewed as performing a path computation which consists of the subcomputations represented by every instruction in the path. Some instruction may be executed and yet produces no effect on the path computation.

To reduce the required analysis after each modification, the following simplifying assumption will be made:

All-Essential Instruction Assumption:

Every instruction on a path affects the overall path computation.

The all-essential instruction assumption implies that if an instruction is executed by t tests, then its subcomputation is used by all t tests. If an instruction J is modified, then all tests which traversed J should be rerun because some change may have occurred to the path computation.

Observe that the all-essential instruction assumption may not hold for all modules. For example, a common programming practice is to place all initializations of variables at the beginning of the program. Some of these initializations may have no effect on a particular execution of the program. For modules which do not satisfy the all-essential assumption, the major effect of making this assumption is to produce a conservative analysis. It is better to be *safe* by including more details rather than leaving out some important ones.

There is another version of the all-essential assumption which applies at the system level. In many cases, although the affected modules and their affected instructions are known, it may require a detailed analysis at the instruction level to determine their impact on the existing tests. To avoid this analysis, the following assumption can be made:

All-Essential Module Assumption:

Every module on a module trajectory affects the overall computation of the module trajectory.

This assumption has similar implications as the all-essential instruction assumption: if a module M is modified, then all integration and system tests which traverse M should be rerun. Similar to the all-essential instruction assumption, this assumption may not hold for all programs. Some modules only pass parameters on to other modules and may not affect the overall computation of a particular execution of the program.

Specification

It will be assumed that during the specification phase, the *software specification* is created and that it gives a precise and unambiguous description of the software functionality. This specification describes what the software as a whole will do without describing the means by which it will be accomplished. The focus is on the software's intended behavior as seen from an external viewpoint. This specification will be used as an oracle for gauging the correctness of the implementation and is the primary input to the design phase.

During the design phase, the software specification is further refined into a *detailed specification*. The *detailed specification* is a set of specifications about the overall program structure, the interfaces and interconnections among modules, and the functionality of each module. The detailed specification includes the *design specification* of each module which gives all input and output or computation relationships of a module. The design specification presents a low-level view of the software that focuses on the functionality of each module. It is highly dependent on the design decisions made during the design phase.

The interdependencies between modules of a program can be formally described using a Module Interconnection Language (MIL) [11]. The first MILs represented intermodule dependencies simply by listing the resources provided by each module, and the resources required by each module. A *resource* is an entity that had a representation in the implementation language. For example, function, type definition and variables can be resources. The interface of a module is characterized by the set of resources it provides and the set of resources it requires. These early MILs all described the module interface using the syntax of the implementation language.

Modern MILs use the notion of *abstract interface specification,* which is similar to the specification of a typical abstract data type [58]. The functional properties of the resources are specified in a nonprocedural, implementation independent language. This language depends on the specification method used. For the purpose of this research, it will be assumed that the module interdependencies are expressed with a language that resembles the early MILs.

Before presenting the model for the design specification, we first define the *input domain* of a module. A given implementation of module M can be characterized by its input domain and the computations for these inputs. An input variable of M is a variable which appears in an input instruction or an input parameter of M. Typically, input variables can be of different data types. Let $I = (y_1, ..., y_m)$ be a vector of input variables of M, Y_j be the domain of the input variables y_j, which is a set of values that y_j can be assigned. The *input domain* of M is the cross product $D[M] = Y_1 \times ... \times Y_m$. A *module input* to M is defined to be a single point y in the m-dimensional input domain D[M].

The design specification of module M describes M at a higher level than the actual implementation. A design specification can be viewed as describing a function, which can be composed of partial functions. Each partial function defines a computation over a subset of the input domain of the module, and can be viewed as a conceptual unit that can be tested independently. Following Richardson and Clarke's model [64], each partial function will be represented by a *subspec*. Each subspec S of module M is described by (C[S], D[S]), where C[S] is the *subspec computation,* and D[S] the *subspec domain*. C[S] specifies the computation to be done or the expected outputs for module inputs in D[S]; D[S] represents the set of input data for which the subspec is defined. A design specification is assumed to consist of a finite set of subspecs.

From the design specification, the *specification domain* of the module M is defined to be the union of the subspec domains, $DS[M] = D[S_1] \cup ... \cup D[S_k]$, where k is the total number of subspecs in M. Any input from outside DS[M] is assumed to be unacceptable to M and M should output an error message. If the implementation is correct, then the input domain D[M] should be the same as the specification domain DS[M].

Any specification language can be used for the design specification. However, in applying our model, the description should be reformulated as subspec domains and their associated computations. In the late stages of the design phase, the design specification is assumed to be written in a language that is similar to a common programming language that includes more abstract operations. For example, the specification language may represent the repetition of a computation by a closed form expression using some high level notations such as summation and user-defined functions.

For the purpose of our experimentation, we use SPA (Specification for Partition Analysis) [63] as the design specification language. SPA is a hybrid language which extends PDL and Ada and combines predicate calculus and procedural constructs. SPA constructs include conditional values, existential and universal quantification, finite summation and product, assertions and encapsulations for abstract data types and the Ada programming language constructs.

An example of the design specification and its implementation is given in Figure 1. Procedure *FindStudentbyName* is one of the procedures in program *StudentDatabase*, which will be studied in detail in Chapter Nine. This program allows the user to create a student database which records the student names, identification numbers and assignment marks, to enter new records and delete old records, to update assignment marks, and to display statistical information about the class marks. We will use various parts of this program to illustrate new concepts throughout the thesis. Procedure *FindStudentbyName* checks whether an entered name is in the existing student database. A pointer to the student record matching the entered name will be returned if such a name exists in the database; otherwise a boolean flag will be set to indicate that the name is not in the database. *NameEnter* is the symbolic name for the input value of the student name to be searched; *Studentlist* represents the list of student records, and *Found* represents the boolean flag indicating whether the entered name is in the student record list.

2.2. Types of Regression Testing

Two types of regression testing can be identified based on the possible modification of the software specification. *Progressive regression testing* involves a modified software specification. Whenever new enhancements or new data requirements are incorporated in a system, the specification should be modified to reflect these additions. The regression testing process involves testing a modified program against a modified specification. Because of the changes to the specification, many existing tests no longer give the correct input-output relation. Consequently, these tests cannot be reused to test the program.

In *corrective regression testing*, the specification does not change. Only some instructions of the program and possibly some design decisions are modified. This has important implications because the specification-based tests in the previous test set should be valid in the sense that they correctly specify the input-output relation. Thus, many existing tests may be used to test the modification. However, because of possible modifications to the control and data flow structures of the program, some existing structural-based tests are no longer testing the previously targeted program components.

Corrective regression testing is often done after some corrective action is performed on the program. Some examples of corrective action include fixing an error that is discovered during system testing, correcting a performance failure that occurs during maintenance, and enhancing the program performance by optimizing its code. This type of change usually involves minor modifications to the implementation.

Corrective regression testing will not include regression testing which is done after correcting a misunderstanding of the specification. If the specification is misunderstood, resulting in incorrect implementation of functionalities, then the correction required will be *major*, that is, some modules may be deleted, and new modules or invocation sequences may be added. In other words, the calling structure of a program may be changed significantly. Corrective regression testing should only involve changes to a few instructions, possibly distributed in several modules. These changes usually do not affect the calling structure of a program.

```
procedure FindStudentbyName
Resource required:
    NameEnter: name to be searched
    Studentlist: a list of student records
    N: Number of records in the student record list
Resource provided:
    StudentPointer: pointer to the student record which matches NameEnter
    Found: a boolean flag showing whether NameEnter is in Studentlist
Case
    NameEnter in Studentlist =>
        StudentPointer = index of the student record which matches NameEnter
        Found = true;
    NameEnter not in Studentlist =>
        Found = false;
endcase;
```

Specification of FindStudentbyName

```
procedure FindStudentbyName(Student: StudentList;
                            NameEntered: string;
                            var Found: boolean;
                            var StudentPointer: integer;
                            CountOfStudents: integer);
var
    index: integer;
begin
    Found := false;
    for index := 1 to CountOfStudents do
        begin
            if (NameEntered = Student[index].Name) and (not Found)
            then
                begin
                    StudentPointer := index;
                    Found := true;
                end;
        end;
end; { end of FindStudentbyName }
```

Implementation of FindStudentbyName

Figure 1. Specification and Implementation of FindStudentbyName

Table 1 lists the major differences between corrective and progressive regression testing. Typically, progressive regression testing is done after adaptive or perfective maintenance, while corrective regression testing is done near the end of the testing phase in the development cycle and after corrective maintenance. Since adaptive or perfective maintenance is typically done at a fixed interval, for example, every six months, progressive regression testing is usually invoked at regular intervals. By contrast, program failures can occur any time and most of them need to be corrected immediately. Thus, corrective regression testing may be invoked after every correction.

Corrective regression testing	Progressive regression testing
• Specification is not changed	• Specification is changed
• Involves minor modifications to code (e.g., adding and deleting statements)	• Involves major modifications (e.g., adding and deleting modules)
• Usually done during development and corrective maintenance	• Usually done during adaptive and perfective maintenance
• Many tests can be reused	• Fewer tests can be reused
• Invoked at irregular intervals	• Invoked at regular intervals

Table 1. Differences between Corrective and Progressive Regression Testing

The same two classes of regression testing can be applied to the unit testing level. Progressive regression testing should be applied to those modules whose detailed specification has been changed. If no change is made to the detailed specification, then corrective regression testing can be applied to the changed modules. In general, a given modification may involve both types of regression testing at different levels. For example, the system level may require corrective regression testing while the unit level may require both progressive and corrective regression testing.

2.3. Principles of Regression Testing

This section lists the underlying principles of our regression testing strategies.

Extensive Reuse Principle:

An economical way of retesting is to reuse as many previous tests and analyses as possible, since this will reduce the effort needed to design new regression tests. Our strategy places heavy emphasis on reusing the previous test plan.

Identical Procedure Principle:

The regression testing process should follow the same procedure as the testing process and the same standard should be applied to both processes. In particular, if the testing process requires storing certain dynamic information, then the regression testing process should also store that information. If the testing process is required to satisfy a certain structural coverage criterion, then the regression testing process should strive to achieve the same coverage measure.

Dynamic Information Principle:

To selectively reuse tests in a test plan, information relating the tests to the program code must be stored, along with the input-output relationships. For example, if the instruction trajectory is stored for each test, this information can be examined to identify whether the program modification has affected any instruction in the trajectory. If the changes do not affect any of these instructions, the test, which can be either specification-based or structural-based, need not be re-executed because it will give the same result. The stored information should allow a test analyst to determine those test results that may be affected by the code changes.

Adequate Testing Principle:

The correctness of software cannot be regression tested into it; the software should be properly tested in the first place. If the software is poorly tested during development, then large regression testing effort will be needed to improve the overall reliability of the software. By applying both functional and structural testing techniques with automated test tools, a software system can be validated to a reliable level where the probability of a failure is minimal. The previous testing should have adequately tested the program so that regression testing can concentrate on testing the modified and affected code.

2.4. Problem Decomposition

The problem of regression testing may be broken down into two subproblems: the *test selection problem* and the *test plan update problem.* The *test selection problem* is concerned with the design and selection of tests to fully test a modified program. A regression testing strategy which reuses previous tests must identify:

(1) Which previous tests to reuse?
(2) What new tests to design?

After the program modification, the test analyst must use two test selection criteria: a *reuse selection criterion,* R_c, for selecting tests from the previous test set and a *new selection criterion,* N_c, for selecting new tests.

We will call the test selection criterion used to generate the previous test set the *original selection criterion,* O_c. This criterion can be any selection criterion such as data flow analysis, functional testing, or a combination of them. A given regression testing strategy can be characterized by its reuse, new and original selection criteria. For example, Harrold and Soffa's strategy uses data flow testing as the test selection criteria for R_c, N_c, and O_c [23]; Yau and Kishimoto use a partition analysis criterion for all three selection criteria [79].

Although it is obvious that the regression testing strategy depends on N_c and R_c, it is less clear that the strategy should depend on O_c. Since O_c provides the set of tests from which R_c will make its selection, O_c indirectly influences the tests that will be chosen by R_c and contributes to the effectiveness of the test selection strategy.

R_c is unique to regression testing and different R_c have been proposed [23, 79]. Traditionally, every test in the current test set is repeated. Thus, the retest-all reuse selection criterion selects all existing tests:

$$R_c^{all}: \{ t \mid t \in T(O_c) \}$$

where $T(O_c)$ represent the current test set created using O_c.

Recently, many proposed regression testing strategies use a reuse selection criterion that is the same as N_c and O_c [23, 79]:

$$R_c^{same} = N_c = O_c.$$

The unstated objective of these strategies is to achieve the same degree of testing of the modified software as the original program.

To achieve the same degree of testing, it is necessary for N_c to *subsume* or equal to O_c. A test criterion TC_1 *subsumes* another test criterion TC_2 if any test set which satisfies TC_1 will also satisfy TC_2. In some sense, TC_1 subjects the software to "more thorough" testing than TC_2. For example, path testing subsumes branch testing, which in turn subsumes statement testing.

It is more cost-effective to use a N_c that is the same as O_c, since a more "powerful" criterion usually requires more effort for test selection. Most of the proposed regression strategies have $N_c = O_c$. Likewise, R_c can be any criterion which subsumes O_c in order to achieve at least the same degree of testing of the modified program as the original one.

The reuse selection criterion of our regression testing strategy is

$$R_c^o: \{t \mid t \text{ executes the affected components or modules } \& \ t \in T(O_c)\}$$

Basically, we make use of the all-essential instruction and the all-essential module assumptions. R_c^o can be shown to subsume structural test selection criteria such as all-branches and all-define-uses. One advantage of R_c^o is that it does not require extensive analysis in selecting the tests. Another obvious advantage is that fewer tests are selected than the retest-all strategy (that is, $|T(R_c^o)| \leq |T(R_c^{all})|$, where $|x|$ represents the cardinality of the set x).

The second subproblem of regression testing is the *test plan update problem* which deals with the management of a test plan as a program undergoes successive modifications. After regression testing, the previous database storing the testing information must be updated to reflect the new testing results. As described in Section 2.1, the test plan includes the following information: test cases and their objectives, dynamic and static components and the achieved structural coverage. Certain old tests will become obsolete and new tests must be added to test the affected codes and *features* of the software. A *software feature* is defined as a specific function in the software system and can be identified from the software specification; it represents a user-perceived functionality of the system and is usually implemented by a group of modules.

The test plan update problem may be defined as follows:

Test Plan Update Problem:

> Given a program P, and its specification S, a test plan T which satisfies the testing criterion A, a program P' which is a modified version of P, and its specification S', generate a test plan T' which satisfies testing criterion A from T, P, S, P' and S'.

Chapter Three

Impacts of Modifications on the Test Set

Before reusing tests from the current test set, the effect of program changes on the test set must be identified. Each test executes some program components and if any of these is modified, then the test needs to be repeated because it may give a different result. Also, if the program changes affect the original purpose of the test, then the test may need to be examined to check its validity. For example, some test may be designed to execute a specific instruction in the program; if this instruction is deleted, then the test may no longer serve any useful purpose.

This chapter first looks at the effect of code changes on the previous test set. Section 3.1 introduces five test classes which can be used to determine the "reusable" tests. A model of the evolution of the test set is also described. An example showing the application of these test classes is given in Section 3.1.3.

A rarely analyzed property of a test set is the redundancy of certain tests. The redundant tests are structural tests which can be removed from the test set without affecting the achieved structural coverage. In Section 3.2, the concept of a *non-redundant test set* is introduced to capture the idea of a test set which contains no redundant tests. A linear algorithm is developed for computing such a test set.

We first give the definitions to be used to describe the implementation changes. A *change objective* for program P is a user requirement which cannot be satisfied by P. This requirement can be an existing requirement which was not implemented or a new requirement added to the system, and can occur during maintenance or the testing phase of the development cycle. Some examples are: adding or deleting a software feature, and improving the efficiency of the program. Some change objectives may involve specification modification, while others, such as those for debugging and *spare-parts maintenance,* normally require only code modification. In *spare-parts maintenance,* an entire module is replaced by another module which implements the same specification and has a "better" implementation. To achieve the change objective, a new version P' of P must be created.

An *instruction change* of an instruction J can be any change to J such as changing the operands, operators, or the computation, the deletion of J, or the addition of a new J, which accomplishes a change objective. A *module change* of a module M is either a set of instruction changes of M, the deletion of M, or the addition of a new M, which accomplishes a change objective. A *program change* is a set of module changes which accomplishes a change objective. Each change objective may involve many module changes, which in turn may involve many instruction changes. A *code change* will be used to refer to either a module change or an instruction change.

Some code changes also imply a specification change. Any change to a call instruction will involve a specification change to the called module because either the input parameters, output parameters, or both are changed; consequently, the input/output specification of the called module is modified.

3.1. Test Classes for Regression Testing

A code change may affect the validity of tests in the previous test set. Some impact of the changes may be identified from the relationship between the source code and the actual changes; others must be determined dynamically by executing the program. This section first gives the test classification for progressive regression testing and then a simpler one for corrective regression testing.

After a modification is made to the specification, tests in the previous test set can be classified into the following mutually exclusive classes. Note that if a specification is modified, then the program components implementing the specification must also be modified.

(1) *Reusable tests* (RuT) - This class includes both specification-based and structural-based tests. RuT test the unmodified parts of the specification, and their corresponding unmodified program components. Observe that any tests which test the unmodified parts of the specification will remain valid. Reusable tests need not be rerun because they will give the same results as previous tests. Although reusable tests will not be used to test the modification, they should nevertheless be kept in the test set because they can be used for testing future modifications.

(2) *Retestable tests* (RtT) - This class includes both specification-based and structural-based tests which should be repeated because the program components being tested are modified, although their specification has not been affected. Observe that although these tests specify the correct input/output relations, they may not be testing the same program components as before the modification.

(3) *Obsolete tests* (OT) - This class includes specification-based and structural-based tests that can no longer be used. There are at least three ways that a test may become *obsolete:*

   (a) Any test which identifies an incorrect input/output relation due to a modification to the specification is obsolete.

   (b) Any test which is designed to test a certain program component and is not testing the same component due to changes in the software is obsolete. If the targeted program component has been modified, some tests may correctly specify the input/output relation, but may not be testing the same component. For example, in domain testing [77], tests are derived to test the borders of each domain. If some predicate in a program is modified, then some domain borders may be shifted. Although some tests for the old borders still specify the correct input-output relation, they no longer test the new borders effectively.

   (c) Any structural test which does not contribute to the structural coverage of the program is obsolete. Since all structural tests are designed to increase the structural coverage of the program, any structural test which does not increase the coverage measure can be deleted during the testing phase.

Extensive analysis may be needed to distinguish retestable tests from obsolete tests. The term ***unclassified tests*** will refer to tests which may either be retestable or obsolete. After the modification, two new test classes may be introduced:

(1) *New-structural tests* (NsT) - This class includes structural-based tests that test the modified program components. They are designed to increase the structural coverage of the program.

(2) *New-specification tests* (NpT) - This class includes only specification-based tests. These tests exercise the new or changed code implementing the affected part of the specification.

Corrective regression testing can be viewed as a special case of progressive regression testing where the specification is not modified. It is easy to show that corrective regression testing may involve reusable, retestable, obsolete and new-structural tests. There is no new-specification test in corrective regression testing because the problem specification is not modified. Table 2 summarizes the change relationships between test class, specification, and *target component*. The *target component* denotes the program components exercised by the tests. For example, if the target component and its specification are unchanged, tests for this component are reusable; if the target component is changed, but its specification is unchanged, tests for this component are retestable.

A key step in regression testing is to identify the test classes in the current test set. This would allow the test analyst to retain certain tests, rerun some other tests, and ignore (or discard) the rest of the tests in the test set. As described in Chapter Two, different regression testing strategies use different reuse selection criteria to identify the reusable, retestable and obsolete tests. Recall that our reuse selection criterion is based on the all-essential instruction and all-essential module assumptions. We will show how to apply this reuse selection criterion to unit, integration and system testing in Chapters Four, Five and Six, respectively.

Test Class	**Specification**	**Target Component**
reusable	unchanged	unchanged
retestable	unchanged	changed
obsolete	unchanged	changed/ removed
	changed	unchanged/ changed
new-structural	unchanged/changed	new
new-specification	changed	new/ unchanged

Table 2. Classification of Tests According to Changes

3.1.1. Evolution of a Test Set

In general, a test set, T, may not include all types and classes of tests. For example, if specification-based tests satisfy the structural coverage criterion, then there need not be any structural-based tests in a test set. The test classes in a test set change with successive modification to the program. An old test set may not include any structural-based tests, but the new test set may have them due to changes to certain program components. Let T_{OLD} and T_{NEW} denote the test set before and after the regression testing, respectively. For corrective regression testing,

$$T_{OLD} = RuT \cup RtT \cup OT$$
$$T_{NEW} = RuT \cup RtT \cup NsT.$$

For progressive regression testing,

$$T_{OLD} = RuT \cup RtT \cup OT$$
$$T_{NEW} = RuT \cup RtT \cup NsT \cup NpT.$$

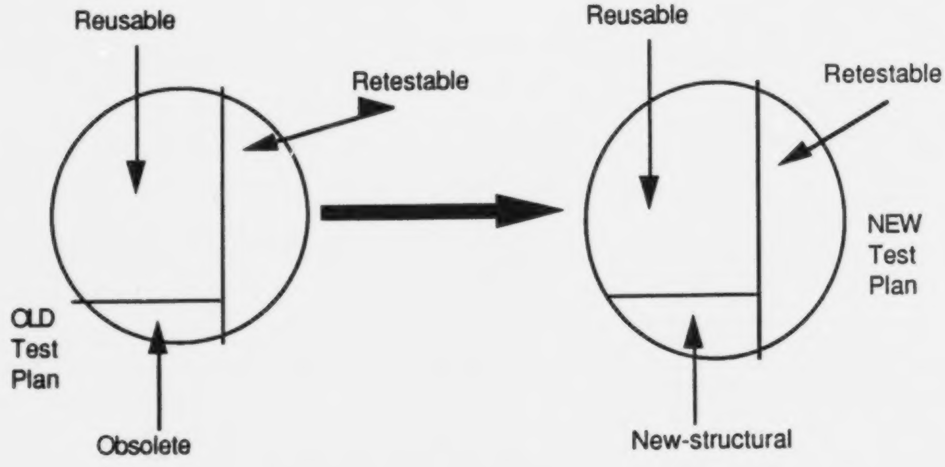
The evolution of a test set under both types of regression testing is illustrated in Figure 2. The total number of tests in a test set may change with each application of regression testing. It should be noted that the specification domain of a module may also change due to modifications occurring in the specification. For example, the specification may change from accepting input of positive integers to accepting input of all integers.

3.1.2. Levels of Use of Tests

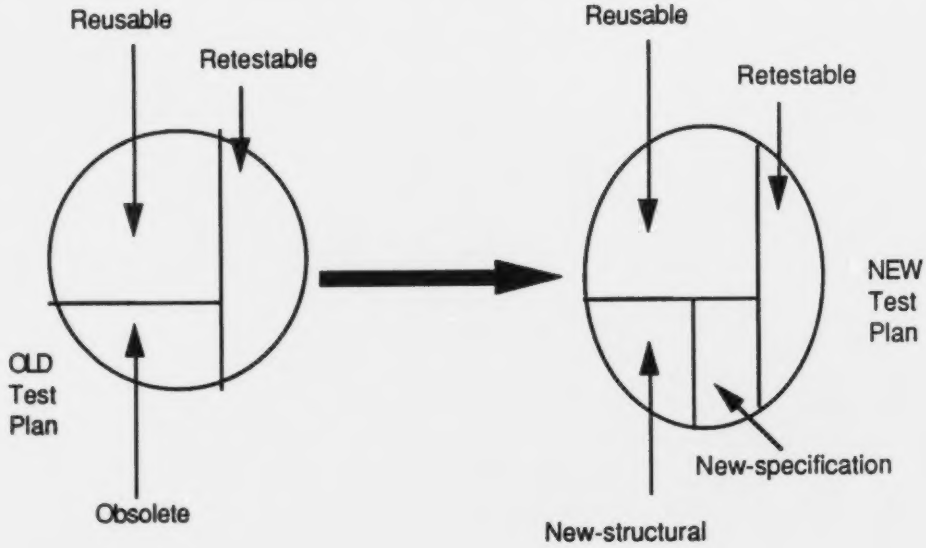
The difference between the reusable and retestable tests may be attributed to different levels of reuse. This section identifies three levels of use for a test. A test can be characterized by its input data, output values and trajectory. Any combination of these components of a test may be changed. A trajectory will be considered changed if any instruction in the trajectory is modified even though it contains the same sequence of instruction identifiers. Also, a new test is assumed to be created whenever changes are made to the input of an existing test.

Based on the above model, the allowable changes to a test are output change, trajectory change or both. It is impossible that the output will be changed without a change in the trajectory when the input is the same as before. Therefore, there are only two possible set of changes to a test: trajectory change, and both output and trajectory changes.

From the above observation, we can identify three *levels of use* of a test. The *first level of use* is to use only the input of the test. In many cases, the testing objective is to exercise some required program components. If the trajectory of an existing test indicates that the required program components are traversed before the changed instructions, then the test input can be used to exercise the same program components. Due to the changes in the program, or the specification, or both, the complete trajectory and the test output may be different from the previous execution. Therefore, tests belonging to the first level of use should be rerun.


Reusable
Retestable
OLD
Test
Plan
Obsolete
Reusable
Retestable
NEW
Test
Plan
New-structural

Corrective regression testing


Reusable
Retestable
OLD
Test
Plan
Obsolete
Reusable
Retestable
NEW
Test
Plan
New-specification
New-structural

Progressive regression testing

Figure 2. Evolution of a Test Set

The *second level of use* is to use both the test input and output. Tests belonging to this level include specification-based tests. When a module has only undergone code modification with its functionalities preserved, then the previous specification-based tests can be used to check the correctness of the implementation. Since the trajectory may be altered or the output may be affected due to the code changes, these tests should be rerun.

The *third (the highest) level of use* is to use the input, output and the trajectory of a test. In this case, none of the instructions in the trajectory of the test is modified. Therefore, the test output will be the same as before. There is no need to rerun these tests.

Tests in a given test set S may be used at different levels. A test set S is *usable at level* i, $1 \leq i \leq 3$, if all its tests are usable at level i or higher. A test set will be given a *usable level 0* if it has at least one test which cannot be used at any level. Observe that a test set at usable level 0 may nevertheless contain some tests that can be used at level 1, 2 or 3.

The relationship between usable levels and the different test classes can now be established. Tests at either the first level or the second level of use correspond to the retestable tests, while those at the third level of use correspond to the reusable tests.

3.1.3. An Example

The program shown in Figure 3 can be used to illustrate the above test classification [62]. This program calculates *x* to the power *y*, with both *x* and *y* integers. *x* and *y* are the only input to the program and their values will be represented by (*x*, *y*) in the following discussion. Suppose the specification-based tests include the following:

(-2,0), (-2,1), (-2,-1), (-1,0), (-1,1), (-1,-1), (0,0), (0,1),
(0,-1), (1,0), (1,1), (1,-1), (2,0), (2,1), (2,-1)

```
        x, y, power : integer
        z, answer : real
 1      read (x,y)
 2       if (y < 0) then
 3            power = -y
        else
 4            power = y
        endif
 5      z = 1
 6      while (power <> 0) do
 7            z = z * x
 8            power = power - 1
        end
 9      if (y < 0) then
10            z = 1 / z
        endif
11      answer = z
12      write (answer)
```

Figure 3. An Example Program

This program has infinitely many infeasible paths. For example, any path that includes instructions numbered (1,2,3,...,9,11,12) is infeasible. Thus, test data cannot be generated for each individual path domain.

Suppose the structural testing criterion is:

(1) exercise all simple feasible paths,
(2) for a loop, iterate the loop one time and iterate the loop at least two times,
(3) if a loop contains several subpaths, tests should exercise all combinations of each pair of these subpaths.

After running the program with the above specification-based tests, the structural-based test (4,2) is added to complete the structural testing requirement. No special test is needed to satisfy criterion (3) because there is only one subpath within the **while** loop. Observe that the set of structural-based tests may be different if a different structural criterion is used.

Consider the following modification. Suppose a faster program is needed. By taking advantage of the binary representation of *y*, a method which performs the computation in time proportional to log *y* can be used. Two extra instructions (6.2 and 8.2) are added to the original program, as shown below. **div** is an integer division operator, and *odd* is a built-in function which returns true if its parameter is an odd integer and false otherwise. After the change, since there is no modification in the problem specification, corrective regression testing should be performed.

```
6      while (power <> 0) do
6.2        if odd(power) then
7              z = z * x
           endif
8          power = power div 2
8.2        x = x ** 2
       end
```

Now the original test set can be classified as follows. Retestable tests include those specification-based tests which traverse the changed instructions: (-2,1), (-2,-1), (-1,1), (-1,-1), (0,1), (0,-1), (1,1), (1,-1), (2,1), (2,-1), and the structural-based test (4,2) because it also traverses some affected instructions. Reusable tests include (0,0), (1,0), (-2,0), (-1,0) and (2,0) because they do not traverse any affected instruction. To satisfy the structural testing criterion, we need to have tests which include paths that traverse all combinations of the two subpaths within the loop. Thus, three new-structural tests are needed: input (2,5) for covering the true branch and then the false branch of instruction 6.2, input (2,7) for covering the true branch of instruction 6.2 two times, and input (2,8) for covering the false branch of instruction 6.2 two times. There are no obsolete tests because the original structural test covers the false branch and then the true branch of instruction 6.2.

3.2. Non-redundant Test Set

The *redundancy* of tests has been ignored by most testing strategies. Programmers usually assume all tests in the test set are essential and rarely remove any of them, although some tests may no longer serve any useful purpose. A *redundant* test is a structural test which exercises the same program components (for example, branches, define-use pairs) as those exercised by another group of tests in the test set. Since redundant tests do not contribute to structural coverage, they should be avoided whenever possible.

A test set is called *non-redundant* if it does not contain any redundant tests. A non-redundant test set T is *minimal* if there does not exist another non-redundant test set T' with |T'| < |T|.

The problem of computing a minimal test set is NP-complete because it can be transformed into the minimal cardinality covering problem [19]. Fischer [16] has been successful in applying zero-one integer programming to solve the minimal test set problem. Recently, Harrold, Gupta and Soffa [26] have developed an heuristic to find an approximation of the minimal test set that has a worst case run-time complexity of $O(c(c+t))$, where c is the number of program components and t is the number of tests.

Although computing a minimal test set is NP-complete, computing a non-redundant test set is algorithmically feasible. We present a solution to this problem below.

There are three advantages in generating a non-redundant test set:

(1) Some tests may be eliminated and this generally will lead to less testing effort during software maintenance.

(2) Each test in the non-redundant test set serves a useful purpose in that it contributes to the coverage measure. Executing any test from the set will increase the coverage measure.

(3) The order of applying the tests in the non-redundant test set is immaterial since they all contribute to the coverage measure. For a redundant test set, the order of executing the tests will affect the final set of tests used to achieve the coverage measure, because some tests may not "add" to the coverage measure if another test is executed first. For example, suppose test *t1* exercises components *c1, c2* and *c3*, and test *t2* exercises components *c1* and *c2*. If t1 is executed first, then when *t2* is executed and is found not to contribute to the coverage measure, *t2* will not be added to the final test set. However, if *t2* is executed first, then when *t1* is executed and is found to increase the coverage, *t1* will be added to the test set. In this case, the final test set will include both tests *t1* and *t2*.

We have developed an $O(ct)$ algorithm for extracting a non-redundant test set from a given test set, where c is the number of program components and t the number of tests. This algorithm computes one of many possible non-redundant test sets, and uses *execution count vectors,* which are columns in the component-test matrix. Let the dynamic behavior of test i be represented by an execution count vector v_i of size c. The j^{th} element of v_i is 1 if the j^{th} program component was executed by test i, and 0 otherwise. The NTS algorithm for computing a non-redundant test set is given in Figure 4. Note that a given set of tests may not cover all the required program components. The variable *covered* is used to identify those components not covered by any tests.

We first show that algorithm NTS only removes redundant tests and then show that the tests remaining each covers at least one program component not covered by other tests. Consider steps 2a and 2b, a test is only removed when $V - v_i$ has no new zero element. In other words, all program components covered by tests i are also covered by the union of the other tests. Therefore test i is redundant and can be removed.

```
Algorithm NTS
Input:      the set of count vectors $v_i$, and the test set
Output:     an non-redundant test set
redundant: a boolean variable denoting that a test is redundant.
covered:    a vector of size c showing whether a program component
            has been covered by a test.
V:          a vector of size c.
begin
        { store sum of all $v_i$ in V, and}
        { mark down which component was not covered }
1       foreach j, j = 1 to c, do
            V[j] = 0
            foreach i, i = 1 to t, do
                $V[j]=V[j]+v_i[j]$
            end
            if V[j]>0 then
                covered[j] = true
            else
                covered[j] = false
            endif
        end
2       foreach i, i = 1 to t, do
            redundant = true
            {for each test i, compute $V=V-v_i$}
2a          foreach j, j = 1 to c, do
                if covered[j] then
                    $V[j] = V[j] -v_i[j]$
                    {if any element of V equal 0, keep test i}
                    if V[j] = 0 then
                        redundant = false
                    endif
                endif
            end
2b          if redundant then
                test i is redundant, remove it from the test set
            else
                {test i should be kept, restore total count}
                foreach j, j = 1 to c, do
                    $V[j] = V[j] + v_i[j]$
                end
            endif
        end
end NTS
```

Figure 4. NTS: Algorithm to Compute a Non-redundant Test Set

We next show that each test covers at least one unique component and it should be kept in the test set. In step 2a, a test is retained if at least one covered element of $V - v_i$ becomes zero. Let the zero element represent program component j. Test i is the only test which covers component j among all the remaining tests. Since we do not take out the counts of those tests remaining in the test set, each test in the final test set has the above property. Observe that the order of analyzing the tests in step 2 may affect the final non-redundant test set.

The complexity of NTS can be determined as follows. Step 1 requires ct additions and c comparisons. Step 2a requires at most c subtractions and $2c$ comparisons for each test. Step 2b requires at most c additions. Since there are t tests, step 2 requires a total of no more than $4ct$ operations, assuming that the comparison requires the same amount of time as the addition and subtraction. Thus, the total complexity of this algorithm is no more than $O(ct)$.

We next illustrate algorithm NTS with the following example.

Let the component-test matrix for a program be as shown in Figure 5. There are 5 program components and 4 tests. The count vectors for the four tests are:

$$v_1 = [01010]^T, v_2 = [01100]^T, v_3 = [10100]^T, \text{and } v_4 = [10111]^T.$$

First, assume the tests are analyzed in their numerical order. $V = [22321]^T$. For the first test, since the new $V = V - v_1 = [21311]^T$ has no 0 element, test 1 is redundant and is removed. For the next step, $V = V - v_2 = [20211]^T$. Because one element of V now becomes 0, test 2 is kept. Carrying out algorithm NTS to completion, we obtain the final non-redundant test set which consists of tests 2 and 4.

If the tests were analyzed in reverse order, the final non-redundant test set would consist of tests 1 and 4.

Program Component	Test			
	1	2	3	4
1	0	0	1	1
2	1	1	0	0
3	0	1	1	1
4	1	0	0	1
5	0	0	0	1

Figure 5. A Component-Test Matrix

If a structural testing strategy aims to maintain a non-redundant test set, it must include a sub-strategy for eliminating redundant tests. It cannot rely on the order of applying the tests to produce a non-redundant test set. The reason is that a new test may render some previous structural-based tests redundant.

As shown above, maintaining a non-redundant test set will require some extra computation. It is up to the test analyst to weigh the benefits of such a test set and the cost of the extra computation.

Chapter Four

Regression Testing at the Unit Level

After the software system is modified, the first step in the regression testing process is to regression test each modified module. This chapter describes a generalized approach which incorporates both functional and structural testing, and this approach can be used with any structural testing strategy such as data flow or control flow structural coverage. Section 4.1 presents the corrective regression testing strategy for a modified module. Section 4.2 gives the corresponding strategy for progressive regression testing, which is a simple extension of the corrective regression testing strategy. Test selection is described in detail in Section 4.2.2.

4.1. Corrective Regression Unit Testing Strategy (CRuT)

The corrective regression unit testing (CRuT) strategy tries to reuse the test set in a way that will reduce the requirement of new tests. The previous test set is used to assist in test selection, and is analyzed in order to classify the various tests; only a subset of the previous test set is rerun. Recall that for corrective regression testing, no new specification-based test is need because the specification is assumed to be unchanged.

As shown in Figure 6, CRuT is divided into two major phases: the *test classification phase* and the *test plan update phase*. The objectives of the test classification phase are to analyze the changes, identify the reusable, retestable, and obsolete tests, and rerun the retestable tests. The test plan update phase updates the test information so that future modifications can utilize up-to-date test information. There is an intermediate step between the two phases which involves developing and running new tests to ensure the structural criterion is satisfied. In Figure 6, module static analyzer and module test repeater together implement the test classification phase and the test plan updater implements the test plan update phase. We next describe each component of the CRuT process.

Module Static Analyzer

Purpose:	To classify the tests and compute the coverage measure.
Input:	Change information.
Output:	Reusable, retestable, and unclassified tests, and the coverage measure.

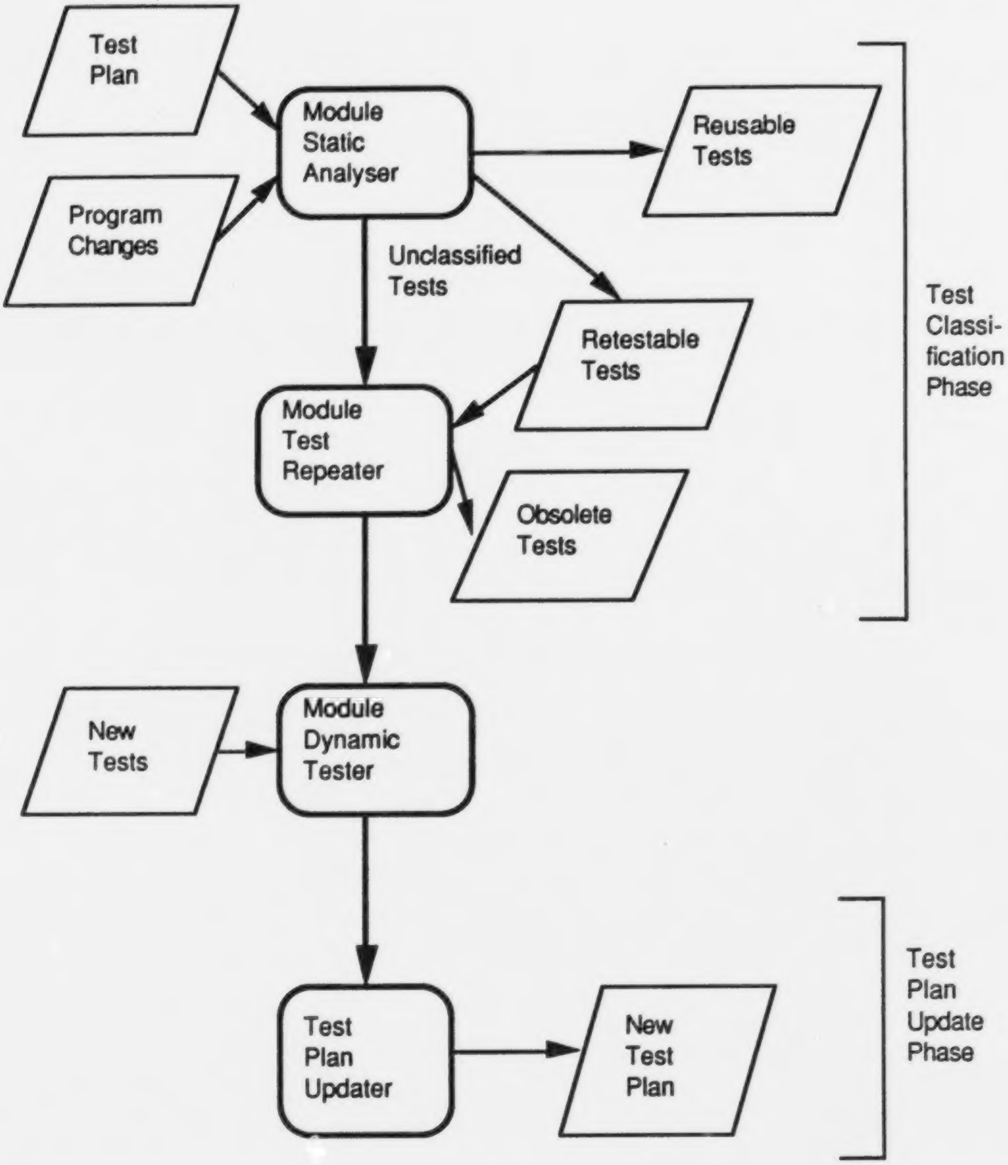

Test Plan
Program Changes
Module Static Analyser
Reusable Tests
Unclassified Tests
Retestable Tests
Module Test Repeater
Obsolete Tests
Test Classi-fication Phase
New Tests
Module Dynamic Tester
Test Plan Updater
New Test Plan
Test Plan Update Phase

Figure 6. The Regression Testing Process of CRuT

This component is *static* in the sense that no test is executed. All analysis is done using the change information and the current database of the test plan. The static analyzer uses this information to group the existing tests into three classes: reusable, retestable, and unclassified. The reusable tests are those tests which do not exercise any modified code; the retestable tests are those specification-based tests which exercise any modified code, and the unclassified tests are those structural tests which exercise the modified code. Another function of the static analyzer is to compute the structural coverage measure achievable from the reusable tests, based on the stored dynamic behavior of these tests. The coverage measure, the retestable and the unclassified tests are passed to the Module Test Repeater.

Module Test Repeater

Purpose: To execute retestable and unclassified tests.
Input: Retestable and unclassified tests, and the coverage measure.
Output: New coverage measure.

This component executes the retestable and unclassified tests in order to

- test the program
- satisfy the structural coverage criterion
- classify the unclassified tests into obsolete and retestable test classes; the unclassified structural tests need to be executed because their trajectory results may change and some of them may not add to the structural coverage.

The Module Test Repeater first executes all retestable specification-based tests and updates the structural coverage with the new trajectory results. Next the unclassified tests are executed in a random order. The order of applying the unclassified tests is not a concern here because the redundant tests will be removed in a separate step before the end of regression testing. Any unclassified structural test which does not increase the structural coverage measure will be put into the obsolete test class. All others are grouped into the retestable test class. The Module Test Repeater stops test execution once the structural coverage criterion is satisfied. Any remaining unclassified tests may be placed into the obsolete class, and handled according to the testing guideline specified in the test plan.

An output of the Module Test Repeater is the coverage measure achieved by both the reusable and the retestable tests. If this coverage measure is not satisfactory, then the Module Dynamic Tester is invoked; otherwise the testing process enters the test plan update phase.

Module Dynamic Tester

Purpose: To execute new tests.
Input: Total coverage measure from Module Test Repeater.

This component is similar to the Module Test Repeater and is invoked only when the structural coverage criterion is not satisfied by both the reusable and retestable tests. A major function of this component is to execute the new-structural tests which are designed to exercise the modified code or to satisfy the structural coverage criterion.

The same test selection strategy used earlier during the testing phase of the development cycle can be used here. Only those tests which increase the structural coverage measure will be stored. Once the structural coverage criterion is satisfied, the NTS algorithm from Chapter 3 is applied to remove the redundant tests. The Test Plan Updater is then invoked.

The major difference between the Module Test Repeater and the Module Dynamic Tester is that the former involves the execution of certain existing tests and the changing and deleting of existing information in the test plan, while the latter involves the execution of new tests and the addition of new information to the test plan.

There are two advantages in executing the unclassified tests before running the new-structural tests:

(1) Some unclassified tests will test the modified code and therefore they can reduce the number of new tests.

(2) Some unclassified tests may be used to assist in test generation. Based on the trajectory results of the unclassified tests, it is possible to adjust the input so that the required program components are traversed. For example, Korel [40] recently applies this idea by combining dynamic data flow analysis, a function minimization method, and program execution for test generation purposes.

Test Plan Updater

Purpose: To update the test plan.
Input: Reusable, retestable, and new-structural tests, and their execution history.
Output: An updated test plan.

This component is invoked by the other three components to create a new test plan from the reusable, retestable, and new-structural tests. The objective is to generate an up-to-date test plan for the next cycle of modifications and regression testing. Most of the information stored in the new test plan have actually been collected by the Module Test Repeater and the Module Dynamic Tester.

4.2. Progressive Regression Unit Testing Strategy (PRuT)

If the specification of the module is affected by the modification, then progressive regression unit testing (PRuT) should be done to revalidate the module. A major difference between PRuT and CRuT is that the former has to deal with the effect of specification changes on the test selection process. As described in Chapter Three, the design specification Su is assumed to be represented by a set of subspecs $\{Su_1, \ldots Su_k\}$, where k is the total number of subspecs. Any specification modification can be viewed as a combination of the following basic modifications:

- adding a subspec
- deleting a subspec
- replacing a subspec.

Before describing the regression testing strategies, we first list the notations to be used in the following analysis in Figure 7.

Notation	Representation
Su	the design specification of a module, $Su = \{Su_1, \ldots Su_k\}$, where k is the total number of subspecs in the module
SM	the set of subspecs to be modified
SD	the set of subspecs to be deleted
SA	the affected subspecs, $SA = SM \cup SD$
SW	the set of new subspecs added to the module
$T_F(Su)$	the set of specification-based (functional) tests
$T_S(Su)$	the set of structural-based tests
$T(Su)$	$T_F(Su) \cup T_S(Su)$
$T_F(Su_j)$	the set of specification-based tests for testing subspec Su_j
$T_F(SM)$	the set of specification-based tests for testing SM,
$T_F(SD)$	the set of specification-based tests for testing SD,
$T_F(SA)$	$T_F(SM) \cup T_F(SD)$
MP	the set of all program paths in the module
$I(Su_j)$	the set of instructions for implementing Su_j
$Ia(SM)$	the set of instructions modified or deleted due to the modification to SM
$Ia(SD)$	the set of instructions modified or deleted due to the deletion of SD
$Ia(SA)$	$Ia(SM) \cup Ia(SD)$

Figure 7. Notations for Progressive Regression Testing

Several observations can be made from the definitions given in Figure 7:

(1) $Su \supseteq SA \supseteq SD, SM$.

(2) $T(Su) \supseteq T_F(Su) \supseteq T_F(Su_j)$ for all subspec Su_j.

(3) Any given instruction J may belong to several $I(Su_j)$.

(4) Ia(SM) may not include all the instructions required to implement SM because the specification modification may be accomplished by changing a few instructions.

We will define the *testing set* TS(J) of an instruction J to be the set of paths which are affected by J. The testing set TS(S) of a set S of instructions $\{J_1, \ldots, J_i\}$ is the set of paths which are affected by the instructions in S.

$$TS(S) = TS(J_1) \cup \ldots \cup TS(J_i)$$

The proportion of reusable tests depends on the modification to the program and its impact on the specification. Table 3 lists the effects of the changes on the previous tests. The first column shows the values of SA relative to Su. The second column lists the different values of the testing set TS(Ia(SA)) relative to MP. If all the subspecs are affected, then all the previous tests have to be re-examined to determine their validity. In the case that only a subset of the subspecs are affected by the modification (SA is a subset of Su), then the previous specification-based tests for the unmodified subspecs should still be valid and can be reused. In addition, the structural tests which traverse those unmodified paths can also be reused.

If the testing set TS(Ia(SA)) includes all the paths in the program, then no structural tests should be reused. If no subspec is affected ($SA = \emptyset$), corrective regression testing can be used to revalidate the module. In this case, all the specification-based tests can be reused and the structural tests which traverse those unmodified paths can also be reused.

Modification		**Effect on Test Plan**
$SA = Su$		no tests can be reused
$Su \supset SA$	$TS(Ia(SA)) = MP$	no reusable structural test, some reusable spec. tests
$Su \supset SA$	$MP \supset TS(Ia(SA))$	some reusable structural tests, some reusable spec. tests
$SA = \emptyset$	$TS(Ia(SA)) = MP$	no reusable structural tests, all spec. tests reusable
$SA = \emptyset$	$MP \supset TS(Ia(SA))$	some reusable structural tests, all spec. tests reusable

Table 3. The Effect of Modification on the Test Plan

4.2.1. A Strategy for Progressive Regression Unit Testing

This section describes the PRuT strategy, concentrating on the case when the functional specification of a module is modified. It is assumed that the functional tests for testing each subspec have been recorded during the initial testing. Thus, only a simple table lookup operation is needed to identify those tests which may be affected due to a modification to the subspec.

The strategy consists of the same two phases (the test classification and the test plan update phases) as that of the CRuT strategy. The objective of the test classification phase is to determine a set of tests which are not required to be re-executed, a set of tests which can be discarded because they are no longer useful, and a set of tests which should be run to test the program changes. There are three major steps in the test classification phase.

Step 1: Determine the valid specification-based tests.

This is accomplished by identifying all the specification-based tests for SM and SD. The remaining tests (the rest of the specification-based tests and all structural-based tests) in the test plan will be used in step 2. New specification-based tests are then created for SM and SW, if any. These specification-based tests are then applied to the module. More details on test selection are described in Section 4.2.2.

Step 2: Classify the tests.

The remaining tests are grouped into reusable, retestable, and unclassified test classes using a similar procedure as that for corrective regression testing. All retestable tests should be executed first. The unclassified specification-based tests should then be applied to the module to revalidate the unmodified functions. The unclassified structural tests should be applied last. Any obsolete structural tests may be removed from the test plan. The reusable tests need not be rerun because they will give the same output as before.

Step 3: Achieve the previous structural coverage.

New-structural tests are added until the structural coverage is satisfied.

The second phase is the test plan update. This phase removes the obsolete tests, adds new specification-based tests for SM and SW, and adds new structural tests for achieving the required coverage to the test plan.

Figure 8 gives the algorithm for PRuT. Basically, PRuT includes a new component Module Spec Tester and makes use of the components of algorithm CRuT.

4.2.2. Test Selection Considerations

In this section, we analyze the test selection process in more detail. A subspec Su_j of a module is *modified* if its domain $D[Su_j]$ is modified, or its computation $C[Su_j]$ is modified, or both. A domain $D[Su_j]$ is modified if it is either *enlarged* (more input points are added to the domain) or *reduced* (some input points are removed from the domain). For example, a domain may be enlarged from accepting only positive even integers to all even integers. A computation $C[Su_j]$ is modified to $C'[Su_j]$ if the output is changed for the same input point (that is, $C'[Su_j(x)] \neq C[Su_j(x)]$, where x is an input point).

We next describe strategies for selecting specification-based tests for various modifications to a subspec. It is understood that the specification-based tests for a subspec may need to be repeated if any instruction implementing the subspec is modified. Assume module M has undergone specification modification. Let the design specification Su of M consist of the following subspecs: $Su_1, ..., Su_k$.

(1) Deleting a subspec Su_i

There are two subcases:

(1a) $D[Su_i]$ is deleted from the module and no input in $D[Su_i]$ is allowed.

For this case, the previous tests for Su_i should be repeated to confirm that module M will not accept such input.

(1b) $D[Su_i]$ is "divided" among the other subspecs.

The input points in $D[Su_i]$ are given a new computation from other subspecs. In this case, some other subspec Su_j is also modified because its domain is enlarged. New tests should be created to check the new computation in $D[Su_i]$. In some cases, the original test input can be reused to test the new computation. However, other cases may require new tests be chosen from $D[Su_i]$ and the computation checked against the modified subspecs. For example, if $D[Su_i]$ is divided among many subspecs, tests should be chosen from $D[Su_i]$ for each such subspec Su_j.

In both cases, we should have tests chosen from the "deleted domain". It is not recommended to concentrate testing on the remaining subspecs alone. Note that the tests for the "deleted domain" need not be kept in the test plan after they are executed.

```
Algorithm PRuT(M: module with its specification modified)
begin
    Module-Spec-Tester(M)
    CRuT(M)         {invoke the corrective regression process}
end PRuT.

Algorithm Module-Spec-Tester(M)
Input:      SM(M):modified subspecs of M
            SD(M):deleted subspecs of M
            SW(M):new subspecs to be added to M
            T: previous unit test plan
Output:     ns_test: specification-based tests
begin
    ns_test = $\emptyset$
    foreach (subspec $s \in SM(M)$) do
        if (s computation is modified) then
            Test = new specification-based tests for the computation
        else if (only domain is modified) then
            Test = new tests for the affected domain
        endif
        ns_test = ns_test $\cup$ Test
    end
    $T = T - T[SM(M)]$
    foreach (subspec $s \in SD(M)$) do
        if (domain of s is deleted) then
            Test = $T_F[s]$
        else Test = union of tests from each subdomain
            divided among the other subspecs
        endif
        ns_test = ns_test $\cup$ Test
    end
    $T = T - T[SD(M)]$
    foreach (subspec $s \in SW(M)$) do
        if (domain of s is new) then
            Test = new specification-based tests
        else Test = $\emptyset$ {this is a complimentary case of other deleted subspec }
        endif
        ns_test = ns_test $\cup$ Test
    end
    Run ns_test
end Module-Spec-Tester.
```

Figure 8. Algorithm for PRuT

(2) Adding a subspec Su_i

There are two subcases:

(2a) $D[Su_i]$ is a new domain added to the module.

In this case, the input points from $D[Su_i]$ were not previously accepted by the module. New tests should be created from $D[Su_i]$ for testing Su_i.

(2b) $D[Su_i]$ is created from the existing $D[Su]$.

In this case, some other subspec Su_j is also modified because its domain is reduced. At least one test should be created from $D[Su_i]$ for each subspec that has given up some input points to Su_i.

(3) Modifying a subspec Su_i

There are three subcases:

(3a) $D[Su_i]$ is unchanged and $C[Su_i]$ is altered.

The same test input from the previous testing for subspec Su_i can be used to check for the correct output.

(3b) $D[Su_i]$ is reduced and $C[Su_i]$ is unchanged.

Tests from the deleted domain should be created to check that the module is not performing the same computation as before. These tests may be chosen from some previous tests if they happen to lie in the deleted domain. This case is similar to Case (1).

(3c) $D[Su_i]$ is enlarged and $C[Su_i]$ is unchanged.

New functional tests chosen from the added domain should be created. This case is similar to Case (2).

Other types of modification can be modeled as a combination of the above basis cases.

Chapter Five

Regression Testing at the Integration Level

Integration testing is an important phase of the software verification process when all the individual modules are combined to form a working program. Testing is done at the module level, rather than at the statement level as in unit testing. Integration testing emphasizes the interactions between modules and their interfaces. A recent study shows that approximately 40% of software errors can be traced to component interaction problems discovered during integration [1]. Most of these detected errors are due to a misinterpretation of the module specifications.

Although many integration strategies have been described, few give any guidelines for actually generating tests. Myers describes six integration strategies: bottom-up, top-down, modified top-down, sandwich, modified sandwich and big-bang testing [56]. Carey and Bendick discuss build testing [6]. Beizer introduces a "mixed bag" strategy which combines bottom-up, top-down, big-bang and build testing [2]. Appendix I gives an overview of these integration strategies.

Bottom-up testing requires the uses of *drivers* which are modules designed to transmit tests to those modules under test. A driver usually reads in tests from a file, calls the module under test iteratively, and compares the responses with the expected outputs. A top-down strategy requires the uses of *stubs* which are simplified modules designed to provide the same responses as the real modules under the same input conditions.

The proposed integration strategies can be grouped into two types: the *incremental* strategies merge a set of modules (usually just one module) at a time to the set of previously tested modules, while the *nonincremental* strategies group all the modules together simultaneously and test them. None of these integration techniques focuses on selecting the tests.

The next section describes our integration testing model. Section 5.2 gives the common errors and faults that may occur in calling another module. A major mistake occurs when the calling module has an incorrect expectation of the called module. Integration testing objectives, which aim at detecting errors not found during unit testing, are described in Section 5.3; a test selection strategy is also presented.

After regression unit testing of the modified software, the next step is to re-integrate the modified modules with the rest of the program. Section 5.4 presents the regression testing strategies for various basic types of modifications; these strategies emphasize reusing the previous analysis and tests. Section 5.5 describes the concept and construction of a *firewall* and shows three interesting properties of program modifications. The firewall notion is introduced to encapsulate all the modules which should be re-integration tested after a modification. In general, it is not necessary to re-integrate all the modules. This result can be used to reduce the retesting effort.

5.1. An Integration Testing Model

To provide an analysis of regression testing for the integration process, a number of additional assumptions (besides those given in Chapter Two) are made which will be applied throughout this chapter:

(1) The *change information* is correct. The change information indicates those modules and specifications which have been modified, and provides descriptions of the new modules and specifications. Note that this assumption does not imply that the modifications are correct. When *specifications* are modified, an extensive analysis is required to ensure that the modification matches the actual intention. The only case where a modified specification is assumed to be correct occurs when the system specification (or the specification of the primary module in the call graph) is modified; in this case, there may be no other documentation or information available indicating this specification to be in error.

(2) During regression unit tests and regression integration tests, since actual modules are in place, there is no need to use drivers and stubs from previous testing. In early development, the test analyst might use drivers and stubs to return the information an incomplete module should provide and to simulate the behavior of that incomplete module. During regression testing, all modules of the system should be available.

(3) There is no recursive calls between modules in the program and therefore the call graph is a tree. Our general results should hold for programs with recursive calls, except for certain change situations involving the recursion, additional testing may be required.

(4) In general, substantial effort may be required to test for errors caused by *global variables* or *common data areas;* since we are analyzing the call graph for the effects of local data flows through formal parameters passing between modules, we will assume that no global variables exist. In other words, there is no *common* or *external coupling* among modules; all modules are *data coupled* [67]. Chapter Seven analyzes the case when this assumption is relaxed to allow the presence of global variables in a program.

(5) The use of *pointers* has to be restricted so as to avoid the same problem as with global variables in assumption (4). Parameters are allowed to consist of *pointers,* and these pointers can address *external tables;* however, there must be a strict separation between *input tables,* which are read-only, and *output tables,* which are write-only. If an external table could be accessed by pointers from within a module, and could be used for both input and output functions, then this could lead to the same testing problems as those encountered with global variables.

We next review the model of module inputs. Each input to a module M can be viewed as a single point in the m-dimensional *input space* Y, where $Y = Y_1 \times \ldots \times Y_m$, and Y_i represents a single input variable, and where m is the total number of input variables. When the designer of a module M develops the design, a number of constraints on the input space must be taken into account. Thus the functionality of M is designed based upon this *subset* of the input space, called the *input domain* D(M). Input points from D(M) will allow M to achieve its desired functionality, whereas points from outside D(M) in the input space should result in an appropriate error message. For simplicity, it will be assumed that only input points in D(M) will cause M to be executed.

It is important to associate practical program data structures with this theoretical view of input domain. The input domain D(M) for module M can consist of:

- values of parameters passed to M by all modules which call M; these parameter values may contain pointers to external input tables, as discussed in assumption (5);
- all data that can be accessed through these pointers passed as parameter values, such as the external input tables; and
- all data that becomes accessible based on authorization codes passed as parameter values.

It should be emphasized that data *internal* to module M (such as tables of constant values) are not contained in D(M).

Useful tests can be created by analyzing the input space of the called module and the potential input from its calling module. In general, the input spaces of a calling module and its called module will be different; in fact, the two modules may have different input variables, and the number of these input variables may be different. Let A be a calling module which calls module B. A subset of domain D(A) will cause the execution of the calling instruction to B. Before A calls B, the input subdomain of A can be mapped to the input space of B; the range of this mapped subdomain of A will be denoted as *map(A, B)*, a subset of the input space of B. This mapping may be one-to-one, many-to-one, or one-to-many, depending upon the relationship between the input variables to A and those to B. Map(A, B) may not contain all possible inputs to B, and some points from map(A, B) may be outside domain D(B). The "difference" between Map(A, B) and D(B) is a good place to look for errors.

Since both D(B) and map(A, B) are subsets of the input space to B, they can be compared with each other. By intersecting these two sets, the input to B can be grouped into three classes:

(1) inputs which are in both D(B) and map(A, B)
(2) inputs which are in map(A, B), but not in D(B)
(3) inputs which are in D(B), but not in map(A, B).

Input from both classes (2) and (3) are good tests because they can indicate errors in the interactions of A and B. Class (2) occurs when A tries to call B with input outside the specification of B. Class (3) occurs when there are input values in the specification of B which are outside the specification of A's use of B. Unfortunately, these two classes of errors are difficult to detect since it is not always possible to effectively compare D(B) and map(A, B). Symbolic evaluation [8] may be used to evaluate map(A, B) and compare it with the symbolic path domain of B, but there are several problems with symbolic evaluation that remain to be solved [38, 55]. These problems include: (i) a high complexity involved in generating a symbolic path expression for paths that call a function or a subroutine, or paths that have complex loop structures, (ii) difficulties in dealing with array and pointer variables, and (iii) a high computational effort required for solving complex path expressions for test data generation.

This chapter shows how to use this model of integration testing in a regression testing mode, where, depending upon the extent of module changes and the interactions between the modified modules and other modules, regression integration testing procedures can be specified.

5.2. Common Errors and Faults in Calling Another Module

In this section, we identify the common errors that may occur when one module calls another module. Although most of these errors should be detected during unit testing, there are some which cannot be detected before integration. These common errors will be classified into two groups based upon whether they can be detected during unit or integration testing, respectively. Let the calling module be denoted by A and the called module by B.

(I) *Interpretation error.*

It can be argued that there are three specifications of a module: First, the *design specification* which is (or created from) the design document, second, the *actual specification* which corresponds to the behavior of the module (reality), and third, the *interpreted specification* as perceived by a user of the module. For simplicity and the purposes of this analysis, it is assumed that the design specification is the same as the actual specification S_a of the module, and must be the source used to judge the correctness of the use of a module. An *interpretation error* is a misunderstanding by the module user about S_a, in that it differs from the interpreted specification S_i.

Since it is difficult to reconstruct S_i, testing should ensure that the user of the module has not made a wrong assumption. A common problem in merging modules together is that the kind of behavior expected by a user of a module may not be the same as the functionality provided by the module. An interpretation error occurs whenever the interpreted specification differs from the actual specification. The implementer of a calling module may misunderstand the functions of the called module or may be given an incomplete specification of the called module. For example, a calling module may wrongly assume that the called module will return a sorted list rather than an unsorted list. In general, there is no testing method to guarantee the detection of this type of error.

There are three classes of interpretation errors: *wrong function, extra function* and *missing function error*. Before presenting these common errors, we first introduce a notation to facilitate the presentation. $map_i(A, B)$ will represent the user perceived $map(A, B)$. In many cases, the actual mapped domain $map(A, B)$ is interpreted incorrectly by the user of module B. The symmetric difference of sets D1 and D2 will be represented by $D1 \Theta D2$.

The mapped domain of module B does not necessarily equal to the domain of B. This difference is represented as $map(A, B) \Theta D(B) \neq \emptyset$. This difference may be interpreted incorrectly by the user of module B. $[map(A, B) \Theta D(B)]_i$ will be used to represent the user perceived difference between $map(A, B)$ and $D(B)$.

(Ia) *Wrong function error.*

The functionalities provided by the called module B may not be those required by the specification of A. The code developer may wrongly assume that B is performing the operation that is needed by module A. In this case, the developer has the correct interpretation between $D(B)$ and $map(A, B)$, that is,

$\text{map}(A, B) = \text{map}_i(A, B)$ and $[\text{map}(A, B) \ominus D(B)] = [\text{map}(A, B) \ominus D(B)]_i$. Thus this error should be detected during unit testing of either module A or B when the same person tests these two modules.

(Ib) *Extra function error.*

Given the specifications of modules A and B, there are some functionalities of B not required for A, and there are some inputs from A which may invoke these functions and cause a failure in A. The developer has not considered other functionalities of B which *may be used* by A. If some inputs cause these extra functionalities of B to be executed, unexpected results may be generated. This error occurs when $\text{map}(A, B) \neq \text{map}_i(A, B)$, and cannot always be detected during unit testing.

(Ic) *Missing function error.*

There are some variable values from A used as input to B which are outside the specification of B; this can be viewed as B failing to supply all the functionalities required by A. Although some functionalities of B are exactly those that are required by A, there are other functionalities required by A that cannot be provided by B. If such functionalities are invoked by A with some inputs to B, unexpected results will be generated. This error occurs when $[\text{map}(A, B) \ominus D(B)] \neq [\text{map}(A, B) \ominus D(B)]_i$, and cannot always be detected during unit testing.

(II) *Miscoded call error.*

A *miscoded call error* is an error which causes the developer to place the *call instruction* at the wrong point in the program. This error may manifest itself as three possible faults:

Extra instruction fault: the call instruction is on a path which should not contain the call.
Wrong placement fault: the call instruction is at the wrong location on the path which should contain the call instruction.
Missing instruction fault: the call instruction is missing on the path which should contain the call.

A new testing method which has received increasing attention is *fault-based testing*. The goal of fault-based testing is to demonstrate the absence of prespecified faults [54]. Examples of fault-based testing include mutation testing [10] and weak mutation testing [31]. The recently introduced RELAY model [65] uses a fault-based criterion for test data selection and guarantees the detection of errors caused by any fault of a chosen class.

Both wrong placement and missing instruction faults may be expensive to detect. To detect a wrong placement fault, it is necessary to check all possible locations where the call instruction may be placed in the path. Data flow information may be used to reduce the number of checking points. The missing instruction fault may involve many tests because it is not, in general, obvious which path is missing an instruction. Therefore, all paths may have to be checked. However, if the anomalous path can be identified, then the amount of testing required will be comparable to that needed for detecting a wrong placement fault.

(III) *Interface error*.

An *interface error* occurs whenever the interface standard between two modules is violated. For example, the parameters may not be in the correct order and they may not be of the right data types, formats, and input/output modes. Although some of these errors may be detected by an "advanced" compiler, one cannot always rely on such a compiler. A more serious problem occurs when the domains of the actual and formal parameters do not match. For example, a module may be designed to process a specific set of input values; if an actual parameter with a value outside this set is transmitted to the module, unexpected results may occur.

Errors (Ia) and (II) (wrong function and miscoded call errors) should be detected during the unit testing of A and B. Although a wrong function error involving B may not be detected during the unit testing of A, this error should be detected during the unit testing of B. Some errors in classes (Ib), (Ic) and (III) (extra function, missing function and interface errors) may not be detected during the unit testing of either A or B because these errors are not directly tested. Thus, their detection should be the primary objective of integration testing. Consider error (Ib): during unit test of module B, we will test all the functionalities of B, but we cannot test whether module A will ever use some of B's functionalities which are not in the specification of A. Likewise, during unit test of A, this error may not be detected. It is obvious that some error in class (III) will not be detected during unit testing of B or A since their interface cannot be thoroughly checked at that time.

5.3. Selecting Tests for Integration Testing

This section describes our test selection strategy for detecting integration errors. Like most other test selection strategies, this strategy does not guarantee the detection of these errors. First we will identify the objectives of integration testing.

A practical objective of each testing phase is to detect errors which are unlikely to be detected by the previous testing phases. Therefore, integration testing should aim at detecting errors that may not be discovered during unit testing. To begin, we will consider integrating only two modules together. When integrating more than two modules, we can integrate them incrementally by adding one new module to the set of integrated modules. From the previous discussion in Section 5.2, we can identify three practical testing objectives when integrating module A with its called module B:

(1) ensure A will not use functionalities that cannot be provided by B (that is, check for missing function errors),
(2) ensure A will not use other functionalities of B if B supplies more functionalities than those required by A (that is, check for extra function errors),
(3) ensure that the interface standards between A and B are preserved (that is, check for interface errors).

We propose to use two types of tests: *interface tests* and *functional tests*. Some of these tests can be created using a subset of the tests applied during unit testing. By analyzing the trajectory of each test, those tests which traverse the call instruction can be identified and reused for integration purposes. A test of module A is said to *traverse* module B if the test input to A can be used to invoke a call of B, and when this occurs, an input to B which is in the set map(A, B) can be obtained.

A new problem is encountered when we want to "reuse" the unit tests of the called module B to test the interactions with its calling module A. This involves generating inputs to A given the inputs to B. Inputs to B are said to be *sensitized* to A if we can find corresponding inputs to A which cause the required inputs to B to be generated. A different version of the problem of sensitizing a test occurs in several other testing methods [5, 37].

Sensitizing a test is a difficult problem, and such inputs to the calling module may not even exist. When a test input to B cannot be sensitized to its calling module A, this indicates a potential error if the functionality of B should be used by A. Further analysis of A is required to ascertain that A will not use this functionality.

5.3.1. Interface Tests

Interface tests aim to check the calling interface between two interacting modules. There are two types of interface tests. The first type should be applied once to each syntactic call and the second type should be applied to dynamic calls. The first type includes tests which check the data type, format and the parameter passing rules of each parameter, and the order and number of parameters. This can usually be accomplished using static analysis.

The second type of interface tests consists of dynamic tests which check the domain of input parameters. These tests will be called *extremal tests,* which are made up with the extreme values of the input variables.

Although the primary benefit of using extremal tests is to detect interface errors, they may also be used to detect a missing function or extra function error. When testing the integration of module A and its called module B, we should execute those extremal tests of A which also traverse module B. These tests may detect a missing function error. Since extremal values of input variables of A do not necessarily give extremal values of input to B, we should have tests which provide the extremal inputs of B. By *sensitizing* the extremal tests of B to the input of A and applying these tests to A, an extra function error may be detected.

5.3.2. Functional Tests

Both functional tests and structural tests should be applied to the module during unit testing. Howden [31] has recently formalized the functional testing method and provided guidelines for selecting functional tests. The test analyst first identifies the functions which are supposed to be implemented by the program, and then selects test data that can be used to check that the program implements the functions correctly.

Any functional tests which traverse a call to another module should be identified and repeated, when the module is being integrated with its called module. These tests are useful for detecting a missing function error. Also, by sensitizing the functional tests of B, we can create another set of tests which may be used for detecting an extra function error. In the case when the specification of A also describes the ways that A will use B, additional functional tests can be created to test specifically for the correct usage of B.

The functional tests of module A will be denoted by f_A. For each instruction J in A, we can determine a subset of f_A which executes J. If J is a call instruction to module B, $f_{A.B}$ will be used to denote functional tests of A which also traverse the call to B. Observe that $f_{A.B}$ is a subset of f_A because they both involve inputs to module A.

For each test in f_B, we may be able to find the corresponding input to A. $f_{B<A}$ is defined to be a set of test inputs to module A with the following property: each test in $f_{B<A}$, when executed by A, will traverse the call instruction of B and invoke B with a set of input parameter values that corresponds to a test from f_B. $f_{B<A}$ is not likely to be a subset of f_B because tests in $f_{B<A}$ require inputs to module A and likely require a different set of input from that for module B.

The integration test set for modules A and B should include $f_{A.B} \cup f_{B<A}$. A problem arises when either some test data from A cannot reach B or when some test data for B cannot be sensitized to A. This does not occur when

$f_{A.B} = f_A$, and
$|f_{B<A}| = |f_B|$.

We have a warning of a possible problem where additional testing and analysis may be necessary if either

(a) $f_A \supset f_{A.B}$, or
(b) $|f_{B<A}| < |f_B|$.

If (a) holds, there are some functional tests of A which do not use B. This implies that there is some function in the specification for A whose effect does not reach B (that is, some function of A does not require any functionality of B). An analysis is required to see whether any input data involving this function would cause B to be invoked; if so, this input should be added to the test set; if not, no further test data are needed. In short, we have to make sure that some input data exercising this functionality do not cause an untested condition to occur in B, the results of which are then returned to A, causing a potential failure.

If (b) holds, there are some functional tests of B which cannot be sensitized to A. This implies that some function in the specification of B may not be used by A. An analysis is required to see whether any input data from A will cause B to be invoked and exercise this function; if so, this input should be added to the test set; if not, no further test data are needed. We have to make sure that some input data from A do not inadvertently cause an untested condition to occur in B, the results of which are then returned to A, causing a potential failure.

5.4. Regression Integration Testing

After a modified module is unit tested, it should be integrated with the rest of the software. The effort involved in integration testing will depend on the extent of the modification and the calling relations between the modified modules and other modules. This section presents a regression testing strategy at the integration level for a set of basis cases. The test selection strategy is described in Section 5.4.2.

There are two types of modification: *non-structural modification* and *structural modification*. Both types of modifications may involve changes to the actual specification of the affected modules. In a *non-structural modification,* there is no modification of the call graph. Observe that enhancing the performance of the system seldom requires modification to the call graph. Also, no structural changes occur during spare-parts maintenance - replacing the entire module by another module which has the same specification and interfaces, but has a "better" implementation.

In a *structural modification,* the edges and nodes of the call graph may be added, removed, or changed. Some possible structural modifications are:

- Adding a new module; an example is to break up a module into two new modules.
- Deleting a module; an example is to merge two modules together.

5.4.1. Independent Instructions

The following analysis requires the notion of the *independent* relation between instructions, which builds on the concept of *scope of influence*. Instruction K is in the *scope of influence* of instruction J if

(1) there is a direct or indirect definition-use relation from J to K, or
(2) if J is a conditional instruction, the execution of K depends on the outcome of J [49].

We will write S=>R to denote that the set R of instructions is in the scope of influence of the set S of instructions.

Instruction K is *independent* of instruction J if K is not in the scope of influence of J and J is not in the scope of influence of K. Instruction J is independent of a set of instructions $S = \{K_1, ..., K_n\}$ if J is independent of each K_i, $1 \le i \le n$. Observe that the instructions in S are not necessarily independent of each other. S || R will be used to denote that the set R of instructions is independent of the set S of instructions.

Figure 9 shows the algorithm for computing the scope of influence to which an instruction belongs. This algorithm consists of three steps. The first step determines, for each instruction i, those conditional instructions which influence the execution of i. This step makes use of a stack variable *scopestack* to store the nested scopes of the conditional instructions. The second step uses any of the well-developed data-flow analysis algorithms [29, 39, 72] to determine the define-use relations. The final step combines the results of the previous steps to give, for each instruction i, a list of instructions whose scope of influence include i.

The computation effort for Algorithm InScopeOf is $O(n^2)$, where n is the number of instructions. Operations such as comparison, union, pop, push, addition and subtraction are assumed to require the same computation time. The first step of Algorithm InScopeOf can easily shown to require $2n$ operations. If we use the data-flow algorithm from [29], the second step requires $O(n^2)$, if the number of basic blocks is the same as the number of instructions. Note that the number of basic blocks is generally less than the number of instructions. Thus, the second step of InScopeOf should require less than $O(n^2)$ in most cases. The third step can be seen to require no more than $n^2/2$ operations. Thus, the complexity of Algorithm InScopeOf is $O(n^2)$.

```
    Algorithm InScopeOf
    n:            total number of instructions in a module
    scopestack:   a stack for storing the current scope
    T:            temporary variable

    begin
        { Step 1: find the scope under conditional instructions }
1       scopestack = ∅
        foreach instruction i, i = 1 to n, do
            inscopeof[i] = scopestack
            if i is a conditional instruction then
                push i onto scopestack
            endif
            if i is an 'endif' or 'end' instruction then
                pop the top element of the scopestack
            endif
        end

        { Step 2: compute define-use relations }
2       Use any dataflow algorithm to identify all define-use relations
        Reformat these relations into the following form:
        foreach instruction i, i = 1 to n, do
            compute df[i] = ($j_1$, ..., $j_h$) if there are direct or indirect define-use
                relations from instructions $j_1$, ..., $j_h$ to i.

        { Step 3: combine the above results }
3       foreach instruction i, i = 1 to n, do
            if inscopeof[i] = ∅ then
                inscopeof[i] = df[i]
            else
                T = df[i]
                foreach instruction j in inscopeof[i] do
                    T = T ∪ df[j]
                end
                inscopeof[i] = T
            endif
        end
    end InScopeOf.
```

Figure 9. Algorithm for Computing Scope of Influence

5.4.2. Basis Cases for Re-integration of Two Modules

This section first describes the integration testing strategies for a non-structural modification, and then for a structural modification. A modification that involves many modules can be viewed as consisting of a combination of basis modifications. Each basis modification involves a pair of calling-called modules, with at least one of them modified.

Given integration strategies for each basis case, we can apply these strategies to any modification.

In the sequel, *NoCh(A)* will denote that module A is not modified, *CodeCh(A)* will denote that module A has undergone code modification but its specification is not modified, and *SpecCh(A)* will denote that module A has undergone specification modification. In most cases, a specification modification also implies a code modification. A special case can occur when the specification of a called module is modified; it is possible that the specification of its calling module will be affected although there is no actual code modification.

There are eight basis cases for a non-structural modification, each requiring various degrees of analysis and effort for test generation. These cases represent all the different combinations of code change and specification change to either or both calling and called modules. In each case, the integration tests should include the set of functional and extremal tests described in Section 5.3. Some cases can be regression tested using many previous tests while others may involve many new tests.

We next describe the integration strategies for each of the basis cases. As described in Section 3.1.2, a test set is given a usable level 3 if all the components (input, output, and trajectory) of every test in the set can be reused. A usable level 2 means that at least the input and output components of every test in the set can be reused; a usable level 1 indicates that at least the input of every test in the set can be reused. Finally, a test set is assigned a usable level 0 if it contains at least one test which cannot be reused. Any test set at usable level 3 does not need to be executed since the input, output and the trajectory will be the same as before.

(1) NoCh(A), CodeCh(B)

Since this case does not involve any specification modification, both A and B's functionalities and domains are unchanged. All the previous f_A can be used at level 3 and f_B can be used at level 2. The former integration tests $f_{A.B} \cup f_{B<A}$ can be used at level 2. Observe that although f_B can be used at level 2, this does not imply that all f_B tests should be rerun. Only a subset of f_B ($f_{B<A}$) and a subset of f_A ($f_{A.B}$) need to be rerun. These tests can be repeated to confirm that the modification does not affect the behavior of B. If these tests give the same result as before, then the interactions between modules A and B are shown to be the same as before the change. Any ancestor of A will not have to be tested because of the modification to B reflected through A.

(2) CodeCh(A), NoCh(B)

Since neither specification is modified, both f_A and f_B should remain valid. f_B can be used at level 3 because module B is not modified, while f_A can be used at level 2 since only the code of A is modified. Because of the modification to A, some program paths to B may be affected. The relation between the changes and the call instruction will affect the required analysis and selection of tests. There are three subcases to be considered:

(2a) Code changes $\|$ Call(B)

Because the code changes and the call instruction are independent, all the previous $f_{A.B}$ should still execute the call instruction and they can be used at level 3 because the specification of A is not modified. Also, $f_{B<A}$ can be used at level 3 since B and the program paths to B are not modified. No execution is needed because the same results will be generated as before.

(2b) Call(B) => code changes
All the previous integration tests should remain valid because there is no change in the subpaths to the call instruction and the specification of A is not modified. These integration tests can be repeated to check the interactions of A and B.

(2c) Code changes => Call(B)
In this case, f_A are still valid, but they may go through different instructions. Therefore, new $f'_{A.B}$ should be created. By the same token, we may need to sensitize new tests $f'_{B<A}$.

(3) CodeCh(A), CodeCh(B)

This case is the same as case (2) except

- the use level of f_B is downgraded to level 2 because the code of B is modified,
- for subcase (a), the previous integration tests should be repeated to revalidate the interactions between A and B.

(4) SpecCh(A), NoCh(B)

This case is different from case (2) because it involves specification changes. Consequently, progressive regression testing of module A is needed, the use levels of the integration tests are reduced, and more new tests need to be selected. Because B is not modified, the previous f_B are usable at level 3. However, new f'_A may need to be created because of the specification modification to A. Since there is a modification to A, we have to consider the relations between the code changes and the call instruction.

(4a) Code changes || Call(B)
Although the code changes and the call instruction are independent, some f_A are modified. Therefore, we should compute new $f'_{A.B}$. $f_{B<A}$ can be used at level 3 since B and the program paths to B are not modified. There is no need to repeat $f_{B<A}$ since they will give the same results as before the modification.

(4b) Call(B) => code changes
$f_{B<A}$ can be used at level 1 because its input will cause the same subpaths to the call instruction to be executed. However, the output of $f_{B<A}$ may be changed due to the specification modification to A. For the same reason, we should create new $f'_{A.B}$.

(4c) Code changes => Call(B)
Because the code changes may affect the call instruction, we should compute new $f'_{A.B}$ from the new f'_A. By the same token, we need to sensitize new tests $f'_{B<A}$.

(5) SpecCh(A), CodeCh(B)

This case is the same as case (4) except:

- the use level of f_B is downgraded to level 2, and
- for subcase (a), $f_{B<A}$ should be repeated to check the modification to B.

(6) NoCh(A), SpecCh(B)

Because of modification to the specification of B, all f_A are usable at level 2 although there is no change to A. It follows that $f_{A.B}$ are usable at level 2. New f'_B may need to be created because of the specification modification to B. From the new f'_B, we can generate the new $f'_{B<A}$. Also for this specific case, as we shall argue subsequently, A should also be unit tested since the specification of B is changed.

(7) CodeCh(A), SpecCh(B)

Since A has only undergone code modification, the previous f_A are usable at level 2. New f'_B are needed to test B because of the modification to B. Since there is a modification to A, we have to consider the relations between the code changes and the call instruction.

(7a) Code changes || Call(B)

Because the code changes and the call instruction are independent, all the previous $f_{A.B}$ should still execute the call instruction and give the correct output. However, because of the specification modification in B, new $f'_{B<A}$ should be created.

(7b) Call(B) => code changes

$f_{A.B}$ can be used at level 2 because the subpaths to the call instruction and the specification of A are not modified. Because of the specification modification to B, new $f_{B<A}$ should be created.

(7c) Code changes => Call(B)

Since f_A may go through different paths, we should compute new $f'_{A.B}$. We may need to sensitize new tests $f'_{B<A}$ because of the new tests in f'_B.

(8) SpecCh(A), SpecCh(B)

This case is similar to integration testing during the development phase. New f'_A and f'_B need to be created. Observe that small changes in the specification may produce large or small changes in the test set; for example, such a change may render all the previous tests obsolete. On the other hand, a major change in the specification may not affect many tests. For example, the specification of A and B may undergo many changes so that they are better "matched" (that is, B is supplying exactly the required functionalities of A). Since the two modules are better matched, less integration testing may be needed after the modification. For all three subcases, new $f'_{A.B}$ and $f'_{B<A}$ tests are needed.

There is one special case to be considered. It is possible that the code modifications only occur in B and the implementation of A is not modified. The specification of A is modified because of the modification to the specification of B. In this case, we can reuse those test inputs of $f_{A.B}$ for the unmodified functionalities of A because there is no change in the paths to B. Nevertheless, we need new $f'_{B<A}$ because of the modifications in A and B.

Observe that the sets f'_A and f'_B usually include some tests from f_A and f_B respectively, because not all the functionalities of a module are normally affected by the modification. It follows that some tests from $f_{A.B}$ and $f_{B<A}$ may also be included in $f'_{A.B}$

and $f'_{B<A}$ respectively. Also, for those basis cases (2), (3), (4), (5) and (7), where there are multiple subcases, and a modification falls into more than one subcase, the union of the subcases would determine the test set to rerun.

The basis cases which affect the call graph structure are described next.

(9) Adding a new called module

Let the new module be denoted by B and its calling module by A. There are two subcases:

(9a) CodeCh(A)

From the unit testing of B, we can obtain f_B. Although A has to be unit tested, there is no change in its functions and f_A can be usable at level 2. Based on the location of B, we can compute $f'_{A.B} \cup f'_{B<A}$ and use this to regression test the modification.

(9b) SpecCh(A)

In this case, A should be unit tested and new f'_A generated. The integration tests should include $f'_{A.B} \cup f'_{B<A}$.

(10) Adding a new calling module

Let the new module be denoted by A, which calls module B. Since B is assumed to be unchanged, no unit testing of B is required. In this case, module A should be unit tested and f'_A generated. The previous f_B can be used for generating $f'_{B<A}$.

(11) Deleting a called / calling module

Let the affected module be denoted by A and the deleted module by B. No integration testing is needed for A and B, and for the case of deleting a called module, A should be unit tested.

Table 4 summarizes the new test requirements and use levels of previous tests under various change conditions. Columns 3 to 6 list the usable levels of each test set. Under the new columns, a Y is used to indicate that new tests should be created.

Given a change situation, Table 4 can be used to determine the use levels of the previous test sets, identify the testing requirement, and estimate the effort required for integration testing. For example, if case (2a) occurs, there is no need to do any integration testing. If case (2b) is encountered, we only have to repeat the previous integration tests and no new tests need to be created. In those cases when new tests are required, they can be selected by following the guideline provided for functional testing [31]. The same procedure for the initial testing can be used to generate these tests.

From Table 4, we can compare the relative impact of different changes on the previous test sets and the effort for regression testing. From the proportion of reusable and new tests, we can rank the basis cases according to the analysis and effort required to obtain the integration tests. The following relations can be established:

{(1)} -> {(2) (3)} -> {(4) (5)} -> {(8)}, and
{(1)} -> {(6)} -> {(7)} -> {(8)}.

		Level of Use				New			
Case	Changes	f_A	f_B	$f_{A.B}$	$f_{B<A}$	f_A	f_B	$f_{A.B}$	$f_{B<A}$
1	NoCh(A),CodeCh(B)	3	2	2	2				
2	CodeCh(A),NoCh(B)								
a	Changes||Call(B)	2	3	3	3				
b	Call(B)=>Changes	2	3	2	2				
c	Changes=>Call(B)	2	3	0	0			Y	Y
3	CodeCh(A),CodeCh(B)								
a	Changes||Call(B)	2	2	2	2				
b	Call(B)=>Changes	2	2	2	2				
c	Changes=>Call(B)	2	2	0	0			Y	Y
4	SpecCh(A),NoCh(B)								
a	Changes||Call(B)	0	3	0	3	Y		Y	
b	Call(B)=>Changes	0	3	0	1	Y		Y	
c	Changes=>Call(B)	0	3	0	0	Y		Y	Y
5	SpecCh(A),CodeCh(B)								
a	Changes||Call(B)	0	2	0	2	Y		Y	
b	Call(B)=>Changes	0	2	0	1	Y		Y	
c	Changes=>Call(B)	0	2	0	0	Y		Y	Y
6	NoCh(A),SpecCh(B)	2	0	2	0		Y		Y
7	CodeCh(A),SpecCh(B)								
a	Changes||Call(B)	2	0	2	0		Y		Y
b	Call(B)=>Changes	2	0	2	0		Y		Y
c	Changes=>Call(B)	2	0	0	0		Y	Y	Y
8	SpecCh(A),SpecCh(B)	0	0	0	0	Y	Y	Y	Y
9	New(B)								
a	CodeCh(A)	2					Y	Y	Y
b	SpecCh(A)	0				Y	Y	Y	Y
10	New(A)		3			Y		Y	Y
11	Delete A or B	0	0	0	0	Y			

Table 4. Breakdown of Tests Used for Regression Integration Testing

Case (1) requires the least effort in generating the regression tests, while case (8) requires the most effort. In general, test selection for corrective regression testing is easier than that for progressive regression testing. We have put cases (2) and (3) in the same group and (4) and (5) in the same group because each pair requires roughly the same effort.

Table 4 may also be used to select the most cost-effective change alternatives if several exist. If several change alternatives involving different set of basis cases are possible, we can use Table 4 to estimate the regression testing effort of each alternative, and select the one likely to require the least testing effort.

To implement this selection strategy, an accurate recording of the functions being exercised by each test must be available. This information can be stored in table form. After the modification, the first step is to trace the specification changes to the affected functions and then examine the table to find tests related to these functions. Note that none of the regression analysis will depend explicitly on the type of integration originally used (see Appendix II).

5.5. The Notion of a Firewall

During regression integration, it is not clear what strategy should be used to integrate the modified modules with the rest of the system. There are two common re-integration strategies. The first strategy tests only the integration of modified modules, which is insufficient because some important module interactions may not be tested. The second strategy repeats the entire integration testing process beginning with the modified modules. This is usually unnecessary and a more cost-effective solution based on the concept of a *firewall* is presented in this section.

A *firewall* is used to discompose the modules of a modified system into two disjoint sets: a "regression set" which includes all modules that should be integration tested and another set consisting of the remaining modules. The firewall is the set of *module invocations* which encloses the regression set and separates the modules into the two disjoint sets. We will show that under certain change conditions, only those modules within the firewall needed be re-integrated and regression tested. Because full integration is avoided, the testing effort can be substantially reduced at no reduction in test effectiveness.

Although the concept of a firewall is fairly simple and intuitive, it has important implications and uses:

- The firewall shows that regression testing all modules is not needed for all cases. If there is no error in the integration of the changed modules and their ancestors or the changed modules with their descendants, then integration testing can stop and there is no need to re-integrate the other modules.
- The size of the firewall gives an estimate of the retesting effort for integration and system testing. It may be used to schedule the regression testing tasks.
- The firewall can be used during system testing to focus testing effort on the area which most likely contain errors.

Before giving a formal defining of a firewall, we first establish three change properties. The construction of the firewall will involve consideration of all those modules which are not modified, but directly interact with the modified modules. These are the direct ancestors and direct descendants of modified modules. As shown in Figure 10, the module-pairs for these modifications correspond to basis cases (1), (6), (2), and (4). In the following formal results, our objective is to prove that all modules in these basis cases must be included as modules within the firewall, but given that certain conditions are met, no other unchanged modules need to be considered, and thus retested.

In this development, we need to consider both outcomes of integration testing these basis cases, one for which no error is detected, and another when an error is detected and must be corrected. To argue formally, it is assumed that both unit tests and integration tests are reliable, in that unit tests guarantee that the computation of a module is functionally equivalent to its specification, and that integration tests guarantee that all integration errors of the types identified in Section 5.2 are detected. Moreover, all errors in the system are assumed to be due to the modification, and residual errors from previous development or modifications have been removed. If the above assumptions are not satisfied, then our regression testing strategy will not detect all errors, especially those which were undetected from the previous testing. If the test analyst suspects that there are residual errors from previous development, he should first subject the unreliable program unit to extensive testing using the testing techniques described earlier, then applies the regression testing strategy presented in this chapter.

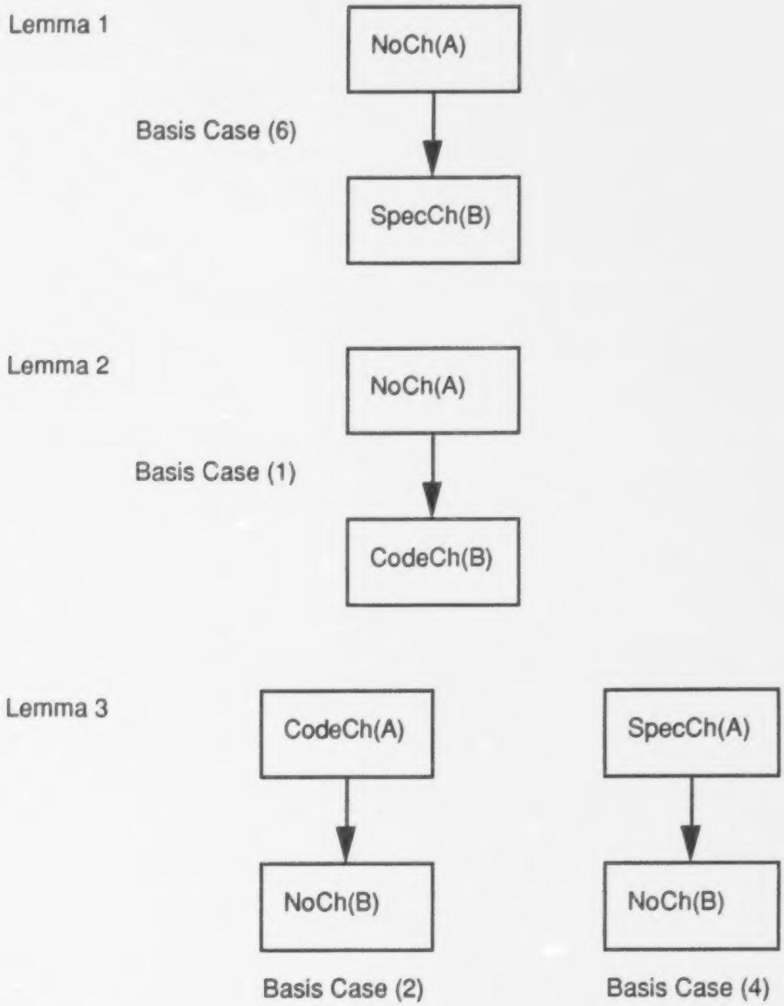

Lemma 1
NoCh(A)
Basis Case (6)
SpecCh(B)
Lemma 2
NoCh(A)
Basis Case (1)
CodeCh(B)
Lemma 3
CodeCh(A)
NoCh(B)
Basis Case (2)
SpecCh(A)
NoCh(B)
Basis Case (4)

Figure 10. Boundary Cases for Firewall Construction

First consider basis case (6) as shown in Figure 10; it is unlikely that this will occur, for ordinarily one would expect at least the code of module A to be modified. Two practical situations might have caused this case to arise:

- only a performance enhancement to B has been made, but no real functional change relative to A, so A can remain unchanged; and

- other modules which call B have required a corresponding change in B; the designer feels that the functionality provided by B to A is still the same.

Lemma 1 shows that if an error is detected, module A may have to be modified after all.

Lemma 1

Consider the basis case (6), where module A is not modified, and A calls module B, whose specification is assumed to have been modified. Assume modules A and B are reliably unit tested, and A-B are reliably integration tested. If an error is detected, then either module A or B (or both) may have to be changed. If no error is detected, then any ancestor of A will not have to be tested because of the modification to B reflected through A.

Proof

First assume that an error is detected: If the error is detected during unit test of A, either A or B will have to be changed (A may have to be code changed). If the error is detected during unit test of B, only B will have to be changed. There are three cases to be considered if an error is detected during integration test of A-B. If the error is an interface error, then it can be corrected in B. If the error is an extra function error, either A or B may have to be changed. If the error is a missing function error, then B can be changed.

Next assume that no error is detected; the unit test of module A ensures that it meets its specifications; the integration test with B ensures that A and B do not have interface errors. Any ancestor of A can be sure that the interface with A is correct, as no changes have occurred. For errors that can be caused by a parameter which "passes-through" module A and interacts with the modified module B, we are sure that they will also be detected. Since this parameter is an input to module A, there must be a function in the specification of A which deals with this parameter, even though the computation with this parameter does not occur until module B (or some descendant of B). This function must be tested during unit test of A, and is also considered during the integration test of A-B, and thus should be correct.

Basis case (6) is unusual in that module A may have to be changed as indicated in Lemma 1; however, this follows because basis case (6) should rarely occur, and the interface above a modified module B should be that described in basis case (1), where only the code of B has been changed. This also provides a guideline for designing the modification, as a code change in a module is preferable to a specification change, to keep the firewall from spreading upward through the call graph.

Next consider basis case (1). We want to argue that when this case is encountered, and it should be encountered in most practical cases, then a "firewall" can be established above the modified modules.

Lemma 2

Consider the basis case (1), where module A is not modified, and A calls module B, whose specification is assumed to be unchanged, but its code is modified. Assume module A is reliably unit tested, and A-B are reliably integration tested. If an error is detected, then only module B has to be changed, and neither the specification nor the code of A have to be modified. If no error is detected, then A or any ancestor of A will not have to be tested because of the modification to B reflected through A.

Proof

First observe that a unit test for A need not be rerun since the specification for modules A and B have not changed. The unit test of B will assure that the functionality in B's specification will be met, but an integration test of A-B is required. If an error occurs in the unit test of B, clearly a further code change in B can correct this. If an error is detected in the A-B integration test, then since the specification and code of A have not changed, the only possibility is that module B must have been invoked. Moreover, since A has not changed, the same parameter values are communicated to B from A. Thus the input-output behavior of B must be different than it was previously, for A has not changed to explain a different output, and this cannot happen given the unit test of B. Since the specification of B has not changed, extra function errors and missing function errors cannot occur; if an interface error has occurred, it can clearly be corrected in module B.

Next assume that no error is detected; the argument that no ancestor of A need be retested because of the A-B interface is the same as Lemma 1, except that we can use the unit test of B to ensure that a parameter "passing-through" module A does not lead to an error in the module which originated that parameter or in any module in between.

Lemma 2 shows that for basis case (1), a "firewall" can be established at A, and since A will not change, no ancestors of A need be examined for testing; it is possible, however, that these modules might be affected through interactions with other modified modules, but not through A.

Next consider basis cases (2) and (4), where a modified module A calls an unchanged module B, and under what conditions descendants of B also need to be examined.

Lemma 3

Consider the basis cases (2) and (4), where module A is either modified in code but not in specification, or its specification is modified, respectively, and A calls B, where B is unchanged. If either the unit tests of A or the integration tests of A-B are in error, then for either case (2) or case (4), we can correct the error by only modifying the code of A. If no error is discovered by either the unit test of A or integration test of A-B, then no descendant of B need be examined or tested because of the change of module A reflected through B.

Proof

An error in either the unit test of A or integration test of A-B is due to an improper design of the new module A. Since the specification of B is unchanged, any detected integration error is due entirely to the change in module A, and so can be corrected in module A alone.

If no error is detected in either the unit test of A or integration test of A-B, then since the specification and code of B are unchanged, no other descendant of B has to be examined or tested. If we are concerned about the effects of a parameter from module A which may be in error, passing-through module B, and affecting some descendant of B, then the unit test of A should suffice to adequately test the functionality of that parameter.

Lemma 3 shows the very serious effect of a design error during maintenance. In the modification of either the code or specification of module A, a common error is to misinterpret the effect of a call to module B during the execution of A. We have shown that it is always possible to correct the error by further modification of A. However, there may be constraints imposed upon such changes, or complexities in such a modification of A. If the designer chooses to modify the specification of B, because it is consider to be "simpler", then one consequence is that now the "firewall" must be extended, as B is changed, and any modules which B calls might also have to be modified in turn if there are errors, thus extending the "firewall" further down in the call graph. Moreover, if any other modules call B, the effect of its change upon their performance must also be established. Thus the potential cost of a design error during maintenance and the decision to modify B rather than simply modifying A is clear.

5.5.1. Constructing a Firewall

Having given the intuitive notion of a *firewall*, this concept can now be formally defined. A graph component is *connected* to a set S of arcs of the call graph if every arc in S is connected to exactly one node of that component. Define a graph component C as all those nodes of the call graph which correspond to modified modules, together with all modules which are their direct ancestors and all modules which are their direct descendants, and all arcs with both nodes in this defined set of nodes.

A *firewall* is composed of the subset E of arcs in the call graph with the following properties:

(1) Graph component C is *connected* to the set of arcs E.

(2) The removal of the firewall arcs E separates graph component C from the rest of the call graph.

Then every module interaction within C must be integration tested while those outside C need not be retested. The function of the firewall E is to clearly separate graph component C from the rest of the call graph.

We next describe a procedure for constructing the firewall. Recall that the call graph is assumed to be a tree. For simplicity, first assume that the set W of all modified modules and arcs between them comprise a connected subgraph of the call graph. Later we will indicate how to handle more complex situations when this constraint is relaxed. The firewall E for W can be constructed as follows:

(a) Initially set E is empty.

(b) Carry out all unit and integration tests using the basis cases within W. If errors are detected between modified modules, then either incorrect specifications or incorrect code should be corrected. For the firewall calculations, we are only interested in the unit and integration tests involving those unchanged modules directly connected to the modules in W.

(c) For each unchanged module A which calls a modified module B in W:

This will likely occur as basis case (1), because basis case (6) is less probable to occur.

If case (6) occurs and no error is detected, then by Lemma 1 there is no need to check the ancestor of A. Add module A, and all arcs from A to modified modules in W, to W. Add all other arcs into and out of module A to the firewall E under construction.

If case (6) occurs and an error is corrected by changing the code of A, then module A is now modified and added by definition to subgraph W together with all arcs from modified modules. All modules connected to A in any way must now be considered for integration testing, and in particular, step (c) must be repeated if A has any ancestors in the call graph.

Lemma 2 indicates that for case (1) even if an error should be detected in the operation of modules A and B, then B can be corrected to rectify the problem and neither the specification nor the code of A need be changed. Thus add module A, and all arcs from A to modified modules in W, to W. Add all other arcs into and out of A to the firewall E under construction.

(d) For each unchanged module B which is called by a modified module A in W:

If no error is detected by unit or integration test, then B can be added to W, together with all arcs from modified modules in W to B. Add all other arcs into and out of B to the firewall E under construction.

If an error is detected involving modules A and B, Lemma 3 assures us that the error can be corrected by changing A. But if the error is corrected by modifying the specification of B, then B is now modified and added by definition to subgraph W together with all arcs from modified modules. All modules connected to B in any way must now be considered for integration testing, and in particular, (d) must be repeated if B has any descendants in the call graph.

(e) After all unchanged modules recursively described in steps (c) and (d) have been considered, the firewall E is complete. If the arcs in E are deleted, the set of modules and arcs in subgraph W form a separate component in the call graph.

In the above procedure for constructing the firewall for integration testing, we assumed that the set W of modified modules and the arcs between them comprised a connected subgraph of the call graph. Because of the distributed nature of computation, and also because several unrelated modifications might be implemented at the same time, this assumption might not always be valid. In this case, W might consist of several connected subgraphs $W_1, \ldots, W_n$, where each consists only of modified modules, but is maximal with respect to the property of being connected.

The only difference between the case where W consists of multiple connected subgraphs and that analyzed in the above procedure is that if the calculated firewalls for two of these subgraphs overlap, then they should be coalesced into one connected subgraph. More specifically, this will occur when a single unchanged module A is called by a modified module in a connected subgraph W_i, and A also calls a modified module in a connected subgraph W_j; then a new connected subgraph should be formed by $W_i \cup W_j$, together with module A and the two arcs involved. For any other connected subgraph W_k, where a modified module in W_k is also connected to module A, then W_k should also be added to this connected subgraph. On the other hand, if an unchanged module calls a number of distinct connected subgraphs $W_1, ..., W_n$, but is not called by any connected subgraph W_k, then these subgraphs should not be coalesced. Note, however, that the unchanged module should be integration tested within each of these connected subgraphs $W_1, ..., W_n$. The situation is similar for the case of a single unchanged module which is called by a number of distinct connected subgraphs $W_1, ..., W_n$.

5.5.2. An Example

Figure 11 shows an example of the use of the basis cases on a given modification to a call graph, and the subsequent calculation of a firewall. The modified modules are labeled A_i, $1 \le i \le 4$, and they form a connected subgraph W. Modules A_1, A_3 and A_4 all have code modification and module A_2 has specification modification. The unchanged modules are labeled U_j, $1 \le j \le 21$. All the basis cases are represented in this example except cases (6) and (8). The firewall E is calculated by first identifying those arcs into and out of unchanged module U_2 which calls a modified module in W. Initially, the subgraph W consists of $\{A_1, A_2, A_3, A_4, (A_1, A_2), (A_1, A_3), (A_2, A_4)\}$. Then U_2, arcs (U_2, A_1) and (U_2, A_3) would be added to W and arcs (U_1, U_2) and (U_2, U_3) would be added to E by step (c). Next we identify unchanged modules which are called by modules in W. If the integration tests yield no errors, then we can add more arcs to E. Specifically, arcs (A_2, U_4), (A_3, U_7), (A_3, U_8), (A_4, U_5), (A_4, U_6), and U_4, U_7, U_8, U_5, U_6 would be added to W and (U_5, U_{20}), (U_5, U_{21}), (U_8, U_{18}), (U_8, U_{19}) would be added to E by step (d). If the arcs in E are deleted from the call graph, W is a component in which all modules must be integration tested. The firewall E is indicated by bold arrows in Figure 11.

5.5.3. Practical Application of the Firewall Concept

In general, the reliable testing assumptions of individual modules and integration errors will only hold for simple programs with straightforward interactions between modules. For programs that cannot be reliably tested, the firewall concept will not guarantee the detection of all integration errors identified in Section 5.2. Nevertheless, this does not diminish the practical uses of the firewall concept because:

(1) The firewall can be used to target testing to the areas which likely contain most errors.

(2) The firewall concept provides a systematic procedure in testing the module interactions. Given limited resources available for testing, this procedure will identify the important interactions to be tested as a high priority.

(3) Since the firewall concept is built on top of the existing testing techniques, it does not preclude the testing of convoluted interactions involving modules within the firewall and those outside. Thus, if the test analyst knows what interactions to test, he can test them. However, if he is not aware of these complex interactions, then applying the firewall concept will not directly test these otherwise unknown interactions.

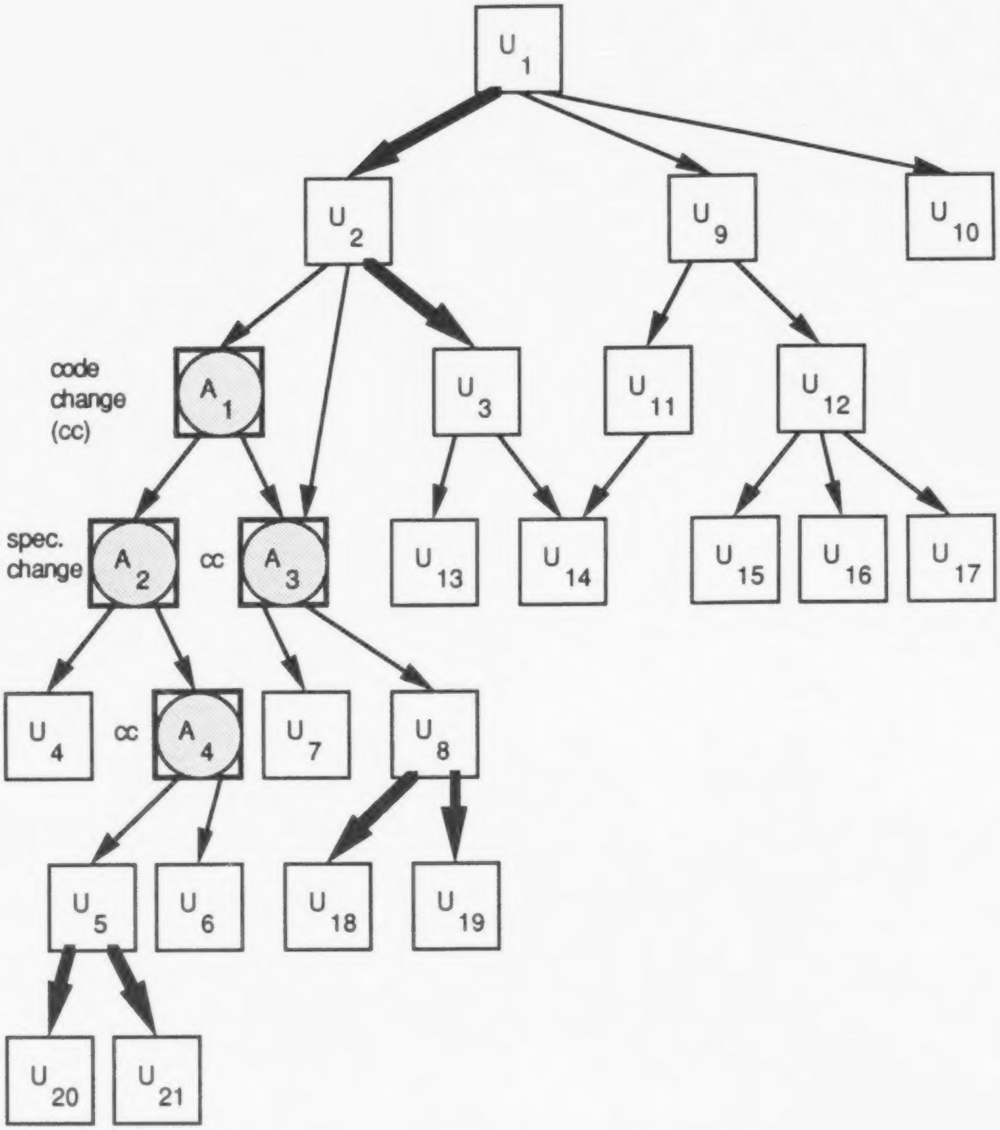

U 1
U 2
U 9
U 10
code change (cc)
A 1
U 3
U 11
U 12
spec. change
A 2
cc
A 3
U 13
U 14
U 15
U 16
U 17
U 4
cc
A 4
U 7
U 8
U 5
U 6
U 18
U 19
U 20
U 21

Figure 11. A Use of Basis Cases and a Calculation of a Firewall

Chapter Six

Regression Testing at the System Level

System testing is the third phase of testing. Before system testing begins, unit testing has established that each module is correct with respect to its own design specification, and integration testing of each pair of calling-called modules has established that the calling module uses the correct functionalities and only the correct functionalities of the called module. If the software is thoroughly unit and integration tested, then system testing may not detect many errors.

System tests have traditionally been created largely based on the specification of the system, and not on its implementation. Black-box testing strategies are commonly used. Some system tests aim to stress the system to uncover its limitations and gauge its full capabilities. These tests should test every possible condition under which the system will be used, including invalid situations to check that the system will give appropriate responses. For example, if the user input is controlled by a menu system, then as many combinations of input options as possible should be executed within the time and resource constraints. A cause-effect graph may be used for test design purposes [14, 57].

Our regression testing strategy for system testing assumes that the all-essential module assumption holds. The implication is that any change to a module is assumed to affect the results of all those previous system tests which traverse the module. Thus, these system tests should be rerun to check the changes.

The objective of regression testing at this phase is the same as the other regression testing phases: achieve the previous level of confidence about the correctness of the software using a reasonable amount of effort. This is accomplished by reusing as many existing tests as possible. In the next section, we present a model for system testing. Most test analysts concentrate on testing as many functionalities of the system as possible, but they do not necessarily use a test termination criterion. Section 6.2 introduces a test coverage criterion which can be used as a termination criterion for system testing. This criterion makes use of the calling hierarchy of the system and the system input to measure the "adequacy" of system testing. Section 6.3 describes our regression system testing strategies.

6.1. A Model for System Testing

This section reviews the assumptions made about the system testing process. The first four assumptions are derived from the test process assumption described earlier in Chapter 2. The last assumption represents a key component of our system testing strategy. The experimentation to be described in Chapter 9 has shown that this strategy is effective in detecting errors and does not require large testing effort.

(1) There exists a mapping between the software features and the software specifications. This mapping will be called the *feature-specification (FS) mapping*. During the design phase, the system specification should be analyzed and each software feature should be mapped to a portion of the specification. The software features should not overlap each other. Those portions of the specification which cannot be mapped to any software feature can be grouped into a feature class called *other*.

(2) Other program information which can be generated during the development phase is the set of dependencies between modules and software features. For both the design and implementation, each software feature can be identified as the set of modules which together implement the feature. This relation is stored in a *feature-module (FM) matrix*. The feature-module matrix is represented by $[FM_{ij}]$, $1 \le i \le f$, $1 \le j \le m$, where f is the total number of feature classes, m the total number of modules, $FM_{ij} = 1$ if module j is a part of the implementation for feature i, and $FM_{ij} = 0$ otherwise. The feature-module matrix can be used by the maintainer to identify the potentially affected modules for a given specification change and the potentially affected software features for a given implementation change.

(3) Since functional or black-box tests are created to test each software feature, we can store the tests for each software feature in a matrix called the *feature-test (FT) matrix*. The feature-test matrix is represented by $[FT_{ij}]$, $1 \le i \le f$, $1 \le j \le t$, where f is the total number of feature classes, t the total number of system tests, $FT_{ij} = 1$ if test j is designed to test feature i, and $FT_{ij} = 0$ otherwise. It is unlikely that there are unique tests for each individual feature. A number of tests are usually required to test a feature effectively, and some of them can also be used for testing other features.

(4) During system testing, a module-test matrix is used to record the dynamic behavior of the program under test. Other component-test matrices such as instruction-test and branch-test matrices may consume too much storage and therefore are not used. Recall that a module-test matrix is represented by $[MT_{ij}]$, $1 \le i \le m$, $1 \le j \le t$, where m is the total number of modules, t the total number of system tests, $MT_{ij} = 1$ if test j traverses module i, and $MT_{ij} = 0$ otherwise. The module-test matrix will be used to identify the tests which should be repeated after changes are made to the implementation. Since the module-test matrix only records the modules traversed by each test, it is impossible to deduce from it the precise effect of a small program change (such as a single instruction change) on the tests.

The set of data structures used for system testing is shown in Figure 12.

(5) System testing will continue until a termination criterion is satisfied. Section 6.2 develops one such criterion.

6.2. A Test Coverage Criterion

In this section, we provide a candidate for a termination criterion for system testing. Most test analysts try to execute all the functionalities of the system but they do not necessarily have a termination criterion. We introduce the *Structural-Interface (SI)* testing criterion which is a hybrid coverage criterion that requires the coverage of a certain set of user input combinations and the coverage of some combinations of modules. Thus, it emphasizes the testing of the user interface and the idiosyncrasies of the implementation at the module level. The SI criterion is simple to compute and is based on the following observations:

(1) The system must be able to respond properly to all the possible combinations of input from the user. When the user is interacting with a system that accepts several different options, he may enter the requested input in any of the possible combinations. For example, a database system may have three user options: insert a record, delete a record and change a record. The user may request an insertion, followed by a change and then a deletion, or any other combination of input requests. Obviously, a complete set of system tests should exercise as many different input sequences as possible. A formal definition of input sequence will be given later when we define the SI criterion.

Feature -	specification Mapping
feature 1	specification a
feature 2	specification b
feature 3	specification c
other	remaining specification

Feature - module Matrix				
Feature	Module			
	1	2	3	4
1	1	0	1	0
2	1	1	0	0
3	0	0	1	1
other	1	1	1	0

Feature - test Matrix					
Feature	Test				
	1	2	3	4	5
1	1	1	0	1	0
2	0	1	0	0	1
3	0	0	1	0	0
other	1	1	1	1	0

Figure 12. Data Structures Used for System Testing

---

(2) A set of system tests which covers all the input sequences may nevertheless fail to exercise many interactions between different modules. This occurs to software systems which are computation intensive or require only a few user inputs. If some interactions between modules have never been exercised, then one cannot have confidence that they will behave correctly if and when they are invoked in future operations. It is likely that there are many possible interactions among the modules,

besides those tested during integration testing. Although testing all interactions may not be practical, one should exercise as many of them as time and computing resources permit.

The *structural-interface criterion (SI)* consists of two subcriteria: the *interface subcriterion* and the *structural subcriterion.* A test set satisfies the SI criterion if it satisfies both the interface subcriterion and the structural subcriterion. The *interface* subcriterion requires the coverage of some input sequences. This criterion can be viewed as "user-oriented" because it emphasizes testing user input combinations and thus provides a certain degree of reliability that the user will not encounter errors during normal operations. This may be viewed as another form of stress testing which emphasizes testing the system under "different" input sequences.

The second part of the SI criterion is the *structural* criterion which is an extension of the structural coverage of unit testing to the system level. In structural testing, tests are selected to traverse some program paths which together exercise the program components (for example, define-use pairs) required to satisfy a structural coverage criterion. Since there are many program paths in a software system, it is not feasible to select tests based on program paths for system testing purposes. One way to reduce the number of tests is to treat each module as a black box and select tests based on the execution orderings of the modules.

The structural subcriterion can be viewed as "implementation-oriented" because it emphasizes testing the idiosyncrasies of the code. It requires the executions of some modules or combinations of them. The required modules can be determined statically with a data structure to be introduced in Section 6.2.2. One motivation for the structural subcriterion is that one cannot have much confidence that the program will behave correctly if the tests do not cause most of the possible module interactions to be executed.

The next two sections provide more formal definitions of the above notions.

6.2.1. Interface Subcriterion

Several interface subcriteria will be developed after we introduce the *input sequence,* the *option set sequence,* and the *input graph.* An *input option* is one value chosen from a set of input values which are *viewed* as *treated similarly* by the program. A set of input is *treated similarly* if the user or test analyst believes that every input in the set will be processed by the program in the same way. For example, a program that classifies an input integer into prime and non-prime may be viewed as having two classes of inputs: prime integers and all the other integers. The user or test analyst may think that all primes will be processed similarly by the program.

A *set of input options* is defined to be the set of *distinct* input options expected by the system at a certain execution point. Input options i and j are *distinct* if they belong to different sets of input values. For example, a database program may prompt the user to select an option from one of the following: (1) enter a student record, (2) delete a student record, (3) update a student record, and (4) exit the system. In this case, there are four distinct input options in this set of input options. Note that some sets of input options may contain only one element. A *null option set* is defined to be an empty set of input options.

An *input sequence (IS)* is a list of input options entered into the system. For each input sequence IS = $o_1 \ldots o_{k-1}$, there is a corresponding *option set sequence (OS)* which is a list of sets of input options $IO_1 \ldots IO_k$ such that o_i is an input option from the set IO_i, $1 \le i \le k-1$, and IO_k is a null option set. There are two kinds of option set sequences: *simple* and *cyclic*. An option set sequence OS = $IO_1 \ldots IO_k$ is *simple* if every IO_i, $1 \le i \le k$, is distinct. An option set sequence OS = $IO_1 \ldots IO_k$ is *cyclic* if there exist some IO_i and IO_j such that $IO_i = IO_j$, $i \neq j$, $1 \le i,j \le k$. An input sequence is *simple* if its corresponding option set sequence is simple. From the previous definitions, a simple input sequence consists of input options chosen from different sets of input options.

The input sequences and option set sequences of a system can be represented by an *input graph*. An *input graph* is a directed graph with a start node S which denotes the first set of input options. Each node with some outgoing edges represents a set of input options, and the *terminal node,* which is a node with no outgoing edge, denotes a null option set. The arrow which connects any two nodes represents an input option selected from the originating node of the arrow. Thus, every arrow in the input graph represents an input option. If the input graph is acyclic, then there is a finite number of *input paths*. An *input path* is an input sequence which traverses the input graph from the start node to a terminal node. An input path is *simple* if its corresponding input sequence is simple. By successively traversing all edges from the start node to each terminal node of an acyclic input graph, we can generate all the input paths.

An input graph may be constructed from a careful analysis of the software, design specification, and the possible user interactions. By first listing all user input at each stage of the system execution and then ordering them based on the dependence of one input on another, we can slowly build up the input graph. The input graph may be viewed as a special case of a state transition diagram, with each state representing possibly a complex sequence of computations and not an instantaneous state of the system. An example input graph is shown in Figure 13. This input graph is actually a subsection of the input graph of Program *StudentDatabase* which will be described in detail in Chapter Nine. Since the input options occur at various modules, to identify where they occur, each node is labeled with the module identifier of the module that includes the input option. To simplify the graph, if a set of input options contains only one input option, the edge representing such an input option will not be labeled.

We next define several interface subcriteria:

(1) *All-option-sets*
Every node of the input graph should be exercised by at least one test. If this criterion is satisfied, then every input option set is exercised.

(2) *All-input*
Every branch of the input graph should be exercised by at least one test. If this criterion is satisfied, then every input option is exercised.

(3) *All-simple-input-paths-and-branches*
Every simple input path and every branch of the input graph should be exercised by at least one test.

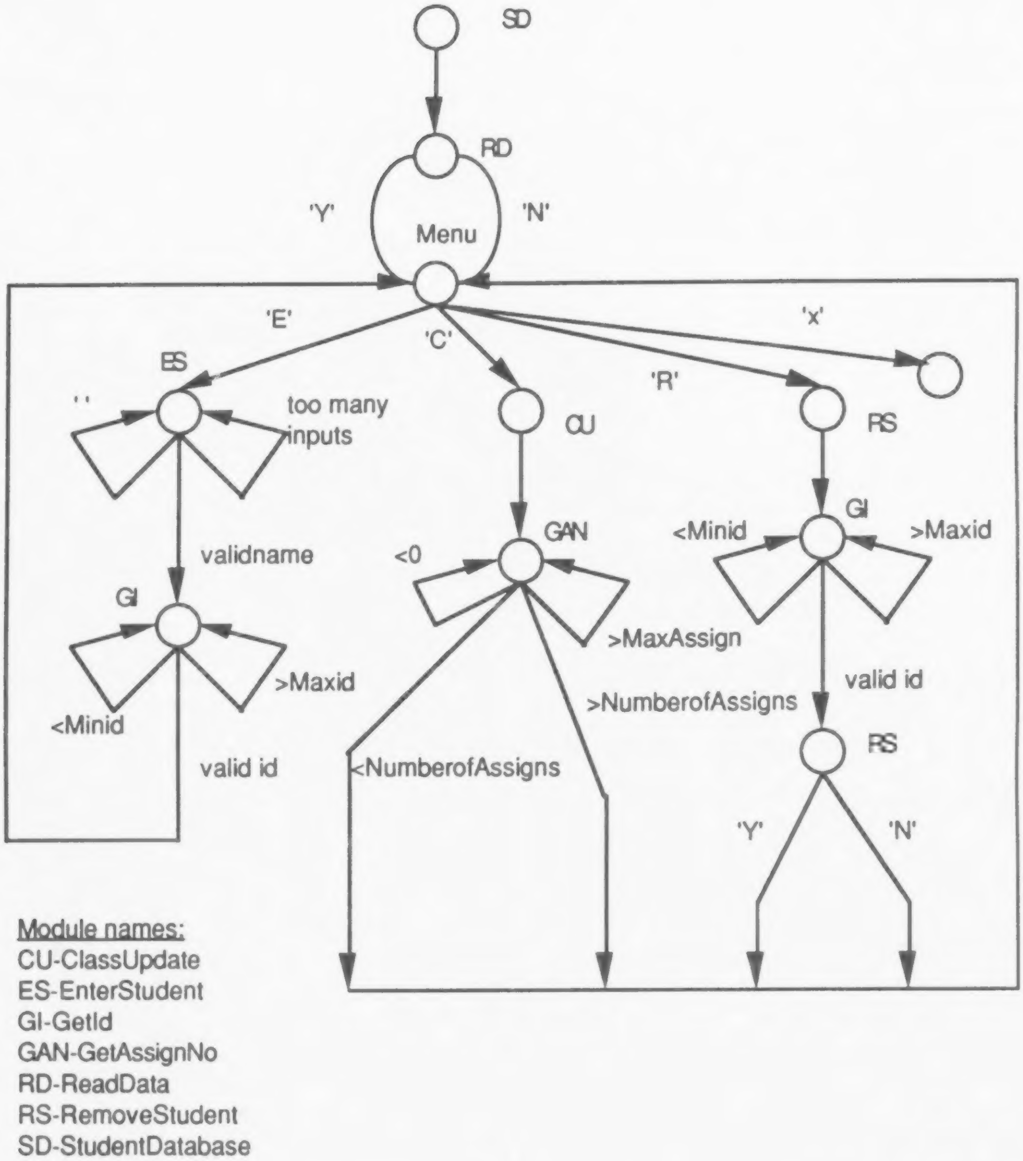

SD
RD
'Y'
'N'
Menu
'E'
'C'
'R'
'x'
ES
' '
too many inputs
validname
GI
<Minid
>Maxid
valid id
CU
GAN
<0
>MaxAssign
>NumberofAssigns
<NumberofAssigns
RS
GI
<Minid
>Maxid
valid id
RS
'Y'
'N'
Module names:
CU-ClassUpdate
ES-EnterStudent
GI-GetId
GAN-GetAssignNo
RD-ReadData
RS-RemoveStudent
SD-StudentDatabase

Figure 13. A Subsection of the Input Graph of StudentDatabase

(4) *Option-set-sequence of length n*
Every option set sequence of length n should be exercised by at least one test, where $n \geq 3$. Since any test set which satisfies the all-input criterion also satisfies the option-set-sequence of length 2 criterion, there is no need to have a criterion for option-set-sequence of length 2.

We will measure coverage of a test set T with respect to (w.r.t.) an interface criterion IC as follows:

$$\text{CoverageI}(T) = E(IC) / N(IC)$$

where E(IC) is the number of required items executed and N(IC) is the total number of required items according to the interface criterion IC. For the all-option-sets criterion, the required items are all nodes of the input graph. For the all-input criterion, the required items are all branches of the input graph. For the all-simple-input-paths-and-branches criterion, the required items are all simple input paths and all branches, and for the option-set-sequence of length n criterion, the required items are all option set sequences of length n. Total coverage w.r.t. an interface criterion is achieved when CoverageI = 1. Note that total coverage may not be attainable because of resource constraints.

It is easy to show that the all-simple-input-paths-and-branches criterion subsumes the all-input criterion which in turn subsumes the all-option-sets criterion. We recommend the use of the all-simple-input-paths-and-branches criterion because (1) the set of tests which satisfies this criterion may be viewed as testing most of the commonly encountered operational scenarios of the system, and (2) it requires a non-trivial number of tests but should require fewer tests than the option-set-sequence of length n criterion.

6.2.2. Structural Subcriterion

Before presenting the structural subcriterion of the SI criterion, we first introduce the *calling order graph* which shows the invocation order of modules for every execution of the system, and then describe a procedure for constructing a calling order graph. The *calling order graph* $Gc = (N, E, n_s, n_f)$ is a directed graph where N is a set of nodes and E is a set of edges in N x N. Each node represents a module, and each edge (n_i, n_j) indicates that module n_i may be invoked *before* module n_j. Note that the existence of the edge (n_i, n_j) does not necessarily mean that n_i calls n_j; the two modules may be called by another module which invokes n_i before invoking n_j. The *start node*, n_s, is the only entry point to the graph and represents the main module, where the execution of the system begins. We assume there exists one exit point, the *final node*, n_f, which is a null node and does not represent any module. A *call path* is a list of nodes $(n_s, n_1, ..., n_k, n_f)$ such that (n_s, n_1), (n_k, n_f), $(n_i, n_{i+1}) \in E$ for all i, $1 \leq i \leq k\text{-}1$. Some call paths may be nonexecutable due to contradictory conditions on the transfer of control from one module to another. A call path is *feasible* if there exists input data which causes the call path to be traversed during program execution. An *atomic* module is a module which does not call another module.

The calling order graph can be constructed from the program text. The construction of the calling order graph is based on the three basic transformations shown in Figure 14. These transformations correspond to the three basic programming constructs of sequencing, alternative and iteration. The ellipses represent undescribed code. There is no call instruction in an ellipsis. In these transformations, Ω represents a null node which is added to make each graph single-exit. Also, every call instruction is represented by one node, even for different calls which invoke the same module. For example, if Call(C) is changed to Call(B) in case II of Figure 14, the calling order graph for A should still have the same structure as shown, that is, one node for each B on each branch. For the sake of presentation, we show each module calls only two other modules. The basic transformations can be generalized to calls to more than two modules.

Figure 15 outlines the procedure for constructing a calling order graph. The first three steps apply the basic transformations and generate a calling order graph that may contain many null nodes. The last step is used to eliminate some of the null nodes.

Figure 16 gives an example of the construction process. The program text outlining the call instructions of each module is shown in Figure 16a. Module A is the main module which calls modules B, C and D. These modules in turn call other modules. By assuming every module is atomic, we first generate the calling order graph of each module, shown next to the respective program text of the four modules. Figure 16b shows the successive substitution of the calling order graphs of modules B, C and D into the calling order graph of A. Finally, the reduced graph is obtained by eliminating the null nodes using the two rules given in Figure 15.

Another example of a calling order graph is shown in Figure 17, which presents the calling order graph of program *StudentDatabase*.

The calling order graph presents a higher level view of the system that does not contain all the details at the source code level. It encodes more information than a call graph and should help the test analyst and maintainer to understand the system. Each call path represents a potential execution of the program. On each call path, modules that are ahead of a module A are invoked or executed before A, and therefore they may affect the computation of A. From the calling order graph, we can develop a family of structural criteria which are suitable for system testing.

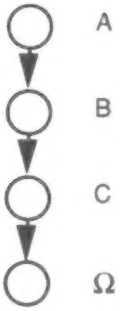
(1) *All-call-nodes.*
Every node in the calling order graph should be exercised by at least one test. If this criterion is satisfied, then every module is exercised at least once.

(2) *All-call-branches.*
Every edge in the calling order graph should be exercised by at least one test. If this criterion is satisfied, then every module invocation is exercised at least once.

(3) *All-simple-call-paths-and-branches.*
Every *simple call path* and every branch of the calling order graph should be exercised by at least one test. A *simple call path* is a path from the start node to the final node in the calling order graph which does not iterate any loop. Each simple call path represents a possible execution of the system and exercising all the simple call paths provides some confidence that most module dependencies have been tested.

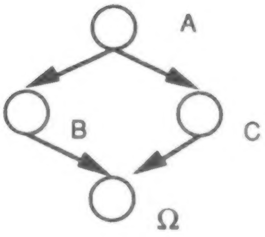
I. Sequencing

```
Module A:
        ...
        Call(B)
        ...
        Call(C)
        ...
```



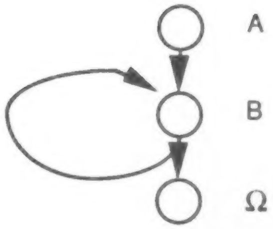
II. Alternative:

```
Module A:
        ...
    if condition1 then
        Call(B)
      else
        Call(C)
        ...
```



III. Iteration:

```
Module A:
        ...
      loop
          ...
          Call (B)
          ...
      endloop
        ...
```



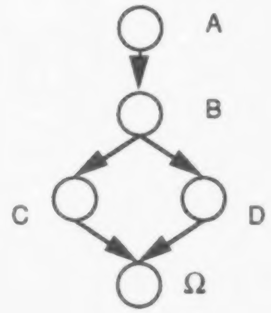
Calling Order Graphs

Figure 14. Constructing a Calling Order Graph

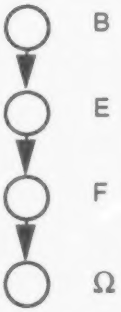
1 For each module i, use the basic transformations shown in Figure 14 to generate a calling order graph for i assuming all its called modules are atomic.
2 For each node in the calling order graph of the main module, replace it with its corresponding calling order graph.
3 Repeat 2 until every node has been replaced by its calling order graph.
4 Reduce the calling order graph by applying the following rules:
  - Two successive null nodes can be combined into one.
  - If a null node is dominated by its immediate predecessor, then combine these two nodes by eliminating the null node.

Figure 15. The Procedure for Constructing a Calling Order Graph

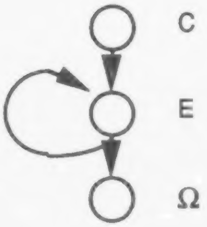
```
Module A:
  ...
  Call(B)
  if cond2 then
    Call(C)
  else
    Call(D)
  ...
```



```
Module B:
  ...
  Call(E)
  ...
  Call(F)
  ...
```



```
Module C:
  ...
  loop
    ...
    Call(E)
    ...
  end
  ...
```



```
Module D:
  ...
  if cond2 then
    ...
    Call(G)
  else
    ...
    Call(H)
  ...
```

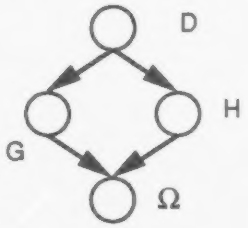


Figure 16a. An Example Construction of a Calling Order Graph

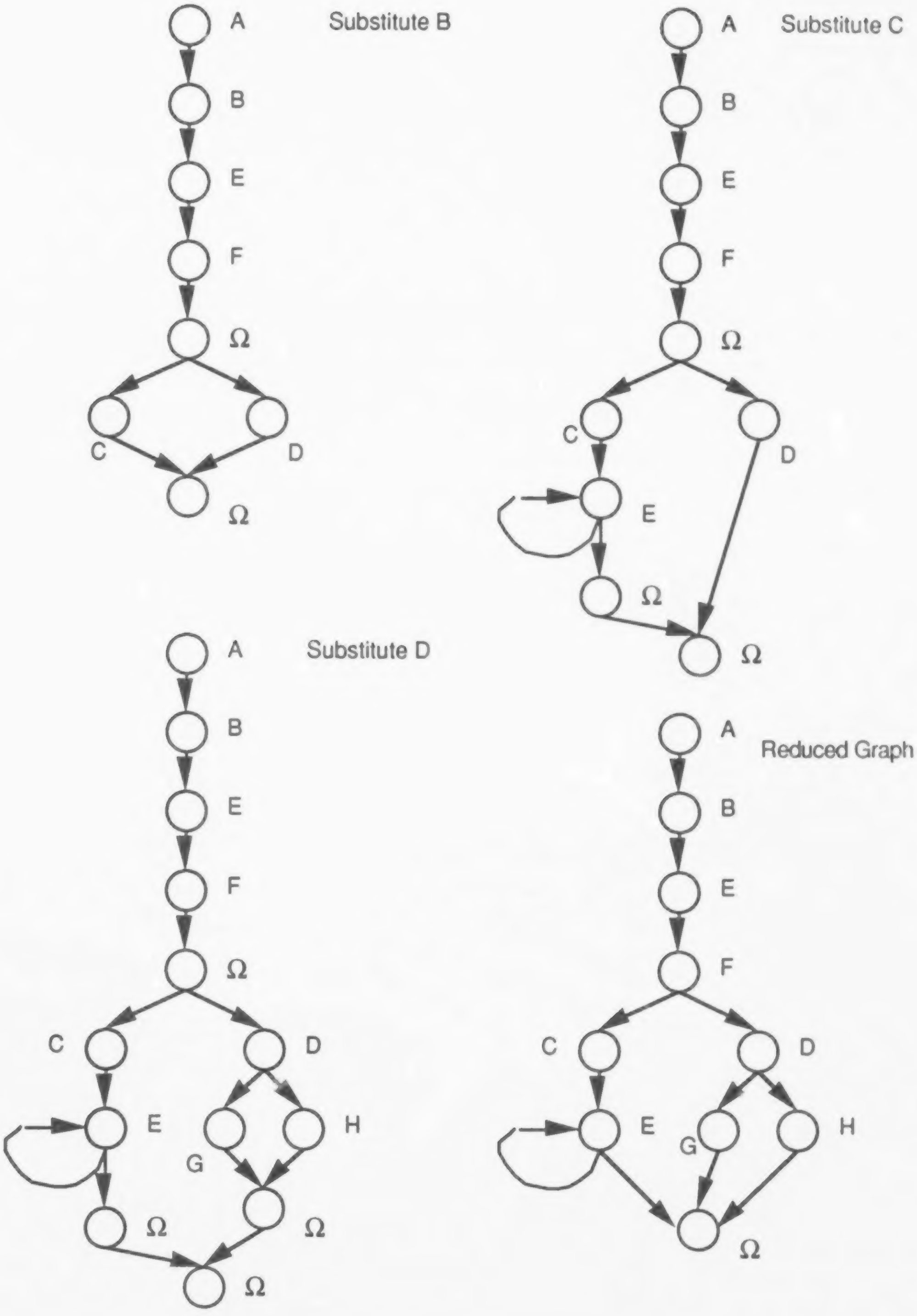

A
Substitute B
B
E
F
Ω
C
D
Ω
A
Substitute C
B
E
F
Ω
C
D
E
Ω
Ω
A
Substitute D
B
E
F
Ω
C
D
E
G
H
Ω
Ω
Ω
A
Reduced Graph
B
E
F
C
D
E
G
H
Ω

Figure 16b. An Example Construction of a Calling Order Graph

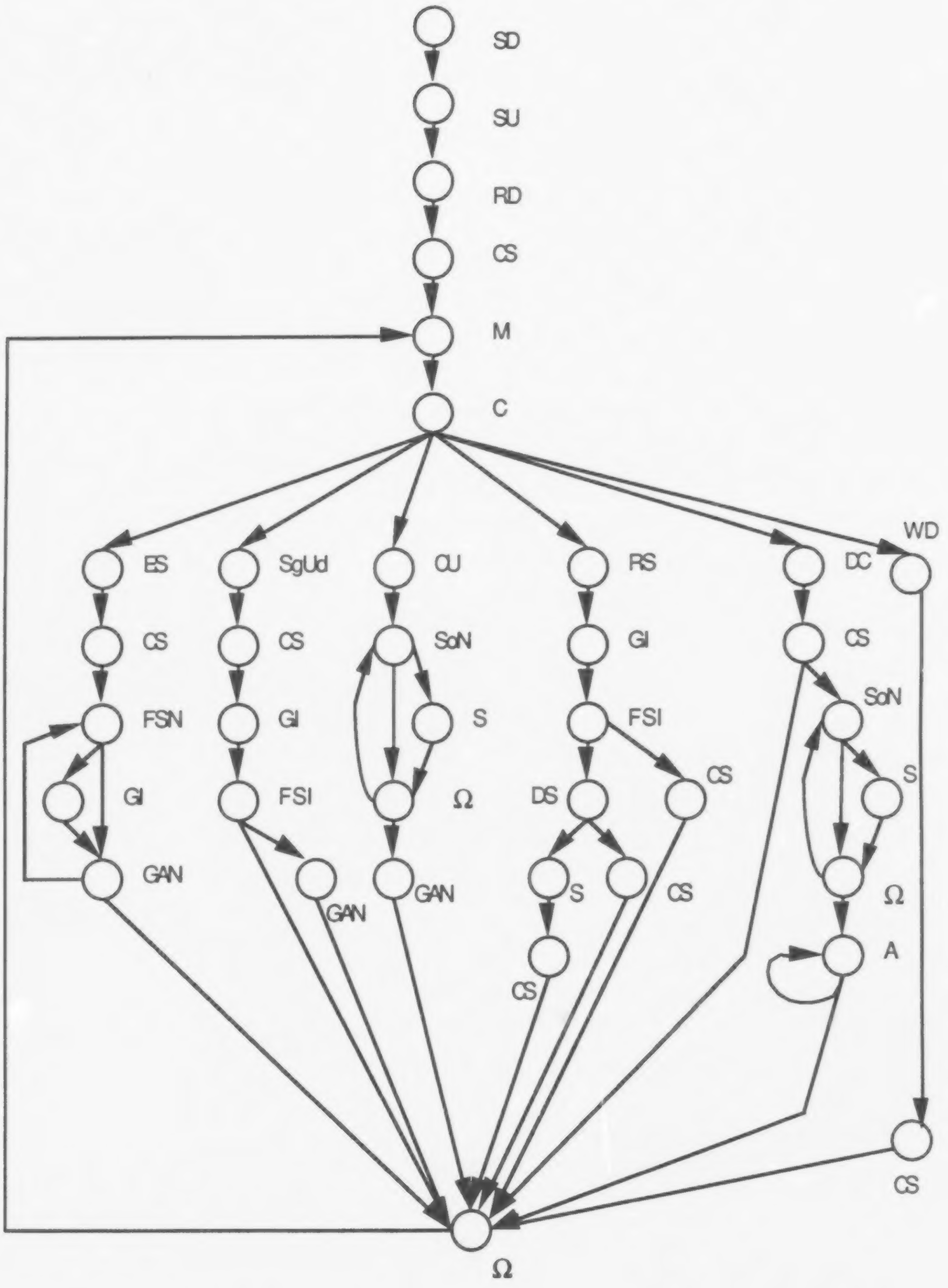

SD
SU
RD
CS
M
C
ES
SgUd
CU
RS
DC
WD
CS
CS
SoN
GI
CS
FSN
GI
S
FSI
SoN
GI
FSI
Ω
DS
CS
S
GAN
GAN
GAN
S
CS
Ω
CS
A
CS
CS
Ω

Figure 17. Calling Order Graph of Program StudentDatabase

We will measure coverage of a test set T w.r.t. a structural criterion SC as follows:

$$\text{CoverageS}(T) = E(SC) / N(SC)$$

where E(SC) is the number of required items executed and N(SC) is the total number of required items according to the structural criterion SC. For the all-call-nodes criterion, the required items are all nodes of the calling order graph. For the all-call-branches criterion, the required items are all edges, and for the all-simple-call-paths-and-branches criterion, the required items are all simple call paths and all branches of the calling order graph. Total coverage w.r.t. a structural criterion is achieved when CoverageS = 1. Note that total coverage may not be attainable because of resource constraints.

One can show that the all-simple-call-paths-and-branches criterion subsumes the all-call-branches criterion which in turn subsumes the all-call-nodes criterion. Since exercising all simple call paths and all branches test more dependencies among modules than the other two criteria, it gives more confidence about the correctness of the system under test than the other two criteria. Also, the number of tests required to satisfy the all-simple-call-paths-and-branches criterion should not be too large and will not require a high test generation effort.

The structural-interface criterion provides a measure of the thoroughness of user input coverage and module coverage that may be used to quantify the program reliability. We will call tests that are created specifically to satisfy the SI criterion the *SI tests*. Given a set T of tests, the *SI coverage* of T is computed as follows:

$$\text{CoverageSI}(T) = (\text{CoverageI}(T) + \text{CoverageS}(T)) / 2$$

When CoverageSI(T) = 1, the test set T satisfies the chosen structural-interface criterion.

6.3. Regression System Testing

Before regression system testing, the test analyst should first collect the "changes" from the maintainer. From the change objective, the maintainer can identify the parts of the specification which are affected, and determines the affected software features, before making the changes to the code. Thus, the maintainer can supply the test analyst with

- the affected parts of the specification
- the set of affected software features F_a
- the set of affected modules M_a, where

$$F_a = F_n \cup F_d \cup F_m$$
$$M_a = M_n \cup M_d \cup M_m$$

F_n is the set of new features, F_d the set of deleted features, F_m the set of modified features, M_n the set of new modules, M_d the set of deleted modules, and M_m the set of modified modules.

Many errors may be made in the process of modifying the software. For example, the maintainer may not identify all the affected software features and affected modules. Thus, besides testing for the modification, a reliable testing strategy should also test the consistency of the information given by the maintainer. Although it is difficult to check whether the maintainer has not missed some change information and therefore supplied an incomplete set of changes to the test analyst, such an omission should be detected by the regression testing strategy.

Similar to the two previous testing phases, there are two types of changes for the regression system testing: code change only and software specification change. We next present the regression testing strategies for these two types of change. Our testing strategy not only tests the modification, but also tests for the consistency of the given information. This strategy does not rely on the maintainer identifying all the changes. It uses the stored information to check whether the maintainer has not overlooked the effect of the changes on all program features.

6.3.1. Corrective Regression System Testing (CRsT)

If the software has been subjected to code change only, then corrective regression testing should be performed. In this case, the test analyst is only given a set of affected modules. Figure 18 gives the procedure for corrective regression system testing (CRsT). There are two sets of tests that can be repeated to test the software system. The first set includes the tests that traverse the modified or deleted modules. These tests can be identified from the module-test matrix.

The second set aims to test the new modules. Observe that some features may have additional module dependency after the change. For example, a module implementing a feature may be split up into two modules; consequently this feature will have a new module dependency.

Some of the system tests may become *obsolete* and may be removed from the test plan. A system test becomes *obsolete* if it was designed to test a feature that has been deleted, or it is a SI test and does not increase the SI coverage. The former class of obsolete tests can be identified from the feature-test matrix; the latter class can be determined by generalizing the algorithm for computing the non-redundant test set to the system testing level. From the module-test matrix, the input graph, and the calling order graph, it is straightforward to identify those SI tests which cover input or calling paths that are covered by other tests. In most cases, no new tests are needed to satisfy the structural-interface criterion.

6.3.2. Progressive Regression System Testing (PRsT)

If the software specification is modified, then progressive regression system testing (PRsT) should be performed. Changes to a software specification are different from changes to a design specification. Some changes to the design specification may not affect the software specification. This may occur for efficiency and maintenance purposes. However, a change of the software specification always induces changes to both the design specification and the program code. Likewise, a change of the design specification always induces some changes to the implementation.

After the modification, the test analyst is given the changes in the specification, the set of affected features, and the set of affected modules. Based on this information, he can identify a set of previous tests to be rerun.

```
Procedure CRsT
input:     t: total number of system tests
           m: total number of modules
begin
    {From the module-test matrix, identify the set of tests $T_u$
    which traverse any modules in $M_m \cup M_d$}
    $T_u = \emptyset$
    foreach module i in $M_m \cup M_d$,
        for j=1 to t,
            if $MT_{ij} = 1$ then
                $T_u = T_u \cup j$
        end
    end
    foreach module in $M_n$,
        Determine its dependency relation with the software features,
        Update the feature-module matrix,
        F = the set of features with new module dependency.
    {From the feature-test matrix, identify the set of tests $T_f$ which test
    any features in F}
    $T_f = \emptyset$
    foreach feature i in F,
        for j = 1 to t,
            if $FT_{ij} = 1$ then
                $T_f = T_f \cup j$
        end
    end
    Repeat all tests in $T_u \cup T_f$.
    Delete obsolete tests.
    Add new tests until the structural-interface criterion is satisfied.
end CRsT
```

Figure 18. The Procedure for Corrective Regression System Testing

There are three basic specification modifications: adding a new software feature, deleting a software feature, and modifying a software feature. In each case, the reusable, retestable, and obsolete tests are first identified. New tests may need to be created to test the affected specification. The regression testing strategies for each case are described below:

(1) Adding a new software feature

This case usually involves adding new modules and modifying some other modules. New specification tests should be created to test the new feature. For the modified modules, the module-test matrix can be used to identify all the affected tests. These tests can then be rerun.

A software feature is *independent* of the rest of the program if it is implemented by its own set of modules and none of these modules is used by other features. An independent feature can be studied and tested alone. In general, if the new software feature is independent of the rest of the program, then regression testing is simplified because none of the previous system tests need to be rerun.

(2) Deleting a software feature

This case may involve deleting some modules and modifying other modules. If the deleted feature is independent of the rest of the program, then some modules should be deleted.

All system tests which traverse the modified modules or are designed to test the deleted feature should be repeated to revalidate the changed software. Although tests which execute the deleted features are obsolete, they should first be rerun and then discarded because they can be used to check that the software indeed will not accept any input which invokes the deleted features.

(3) Modifying a software feature

This case can be viewed as a deletion of a software feature, followed by an addition of another software feature. New tests should be created to test the new feature. Tests which are designed for testing the modified feature should be repeated. Also, those tests which traverse the modified and deleted modules should be rerun. Tests for the latter can be removed from the test plan after testing.

Figure 19 shows the general procedure for progressive regression system testing.

```
Procedure PRsT
begin
    From the changes in the specification, identify the set of affected
        features $F'_a$ from the feature-specification mapping.
    Compare $F'_a$ to the given set of affected features $F_a$.
    if $F_a \neq F'_a$, then
        notify the maintainer about the inconsistency,
        stop regression testing.
    else
        $F = F_m \cup F_d$
        From the feature-module matrix, identify the set of affected
            features $F'$ due to the deletion of modules in $M_d$.
        if not $(F \supset F')$, then
            notify the maintainer about the inconsistency,
            stop regression testing.
        else
            From the feature-test matrix, identify the set of tests
                $T_f$ which test any features in F.
            From the module-test matrix, identify the set of tests
                $T_u$ which traverse any modules in $M_m \cup M_d$.
            Repeat tests in $T_f \cup T_u$.
            Delete entries for $F_d$ and the obsolete tests from the
                data structures (See Note 1).
            For new features,
                add new entries to the feature-specification mapping,
                add new entries to the feature-module matrix.
            Devise new tests for new features and update the
                feature-test matrix. Let this set of tests be $T_n$.
            Run $T_n$ and update the module-test matrix.
            Add new tests until the structural-interface criterion is satisfied.
        endif
    endif
end PRsT
```

Note 1: Any test which is designed to test a deleted feature should be checked to determine whether it also tests other features; if not, this test can be removed from the test plan.

Figure 19. The Procedure for Progressive Regression System Testing

Chapter Seven

Regression Testing Programs Containing Global Variables

A *global variable* is a variable which is referenced by a module other than the one containing its declaration. Using global variables affects not only the understandability, readability, and maintainability of the software, but also its testability. This is especially apparent in integration testing. Global variables create data flow dependencies between modules which are not directly callable and may be well separated in the call graph. For example, in Figure 20, module A defines a global variable g which influences modules B and C in other areas of the call graph. The use of global variables also causes the following problems: (1) it is very difficult to determine the complete effect of a module without looking at its implementation, and (2) the use of global variables may create many maintenance problems because a change in the computation of the global variable may affect many modules and consequently may require the re-analysis of the whole program.

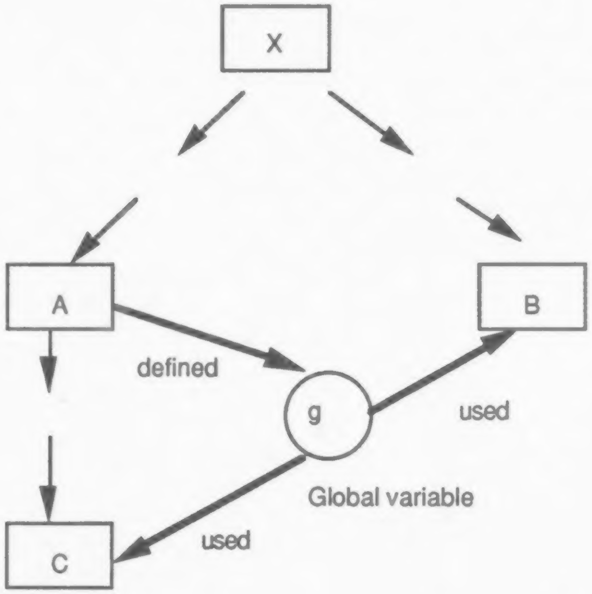


Figure 20. Potential Effect of a Global Variable on Other Modules

Despite the existence of global variables in many software systems, testing global variables has traditionally been overlooked by the testing community. Due to the difficulty in designing tests for global variables, test analysts normally ignore them during the test design and hope that the functional tests will test them indirectly. This state-of-the-practice leads to the release of some software systems with untested properties, and must be corrected to increase the software reliability.

The need for testing global variables can be summarized as follows:

- Global variables are widely used. Although most development methodologies recommend not using global variables, they are in fact widely used in practice. Most programmers use them because of their tendency to take short cuts in programming. Any reliable testing method should test the effects of global variables.
- If global variables are not tested during unit or integration testing, then their effect will not be examined other than by coincidence.

This chapter presents some insights into the problem of testing and regression testing global variables. A basis set of testing problems for parameters will be identified and the testing problem of global variables will be mapped into a combination of these basis cases. It will be shown that global variables can be treated as parameters and can be tested accordingly. The existence of global variables does not introduce additional testing problems other than those similar to testing parameters. Strategies for unit, integration, and system testing global variables are described. Section 7.1 introduces some notations to be used in the sequel and reviews the testing assumptions. Section 7.2 describes the basis cases for testing parameters and they will be used in the presentation of Section 7.3, where we illustrate that global variables can be treated as parameters and tested accordingly. The objective here is not to solve the testing problem for global variables, which is a difficult problem by itself, but to develop a solution to the regression testing problem for global variables. Nevertheless, several observations on testing global variables will be presented in Section 7.4. Section 7.5 describes a strategy for retesting global variables.

7.1. A Testing Model

Before listing the assumptions used in the following analysis, we first introduce some terminology which will facilitate the presentation. A global variable or a parameter is *directly used* in a module M if it is used by an instruction other than a call instruction in M. M will be called a *using module* of the global variable or the parameter. A global variable or a parameter is *directly defined* in a module M if it is defined by an instruction other than a call instruction in M. M will be called a *defining module* of the global variable or the parameter. A call instruction defines a global variable or a parameter if the invoked module assigns a new value to the global variable or the parameter. Since a defining module M of variable v may contain many paths and some paths may not traverse the definition of v, v may not be defined in every execution of M. Similarly, a using module of variable v may not use v in every execution. A global variable or a parameter is *directly referenced* in a module M if it is directly used or directly defined in M.

The definition of a global variable g from a defining module of g may *reach* other modules. A definition of v at module A *reaches* another module B if

- there is a definition-clear path from the definition of v to the exit point of A, and

- there is a module invocation sequence from A to B such that v is not redefined on at least one program path between the definition of v at A and the use of v in B.

A definition of variable v which reaches a module M is *killed* by M if M redefines v on all paths through M.

Note that a using module of a global variable g uses the definition of g from another module. A *define-use module pair* is a defining module A and a using module B of the same variable v such that the definition of v in A may reach B.

There are some modules which just pass the input parameters to another module, and do not directly reference them. Module M is a *transferring module* of variable v if M satisfies the following conditions:

(1) v is an input parameter of M.

(2) M is not a using or a defining module of v.

use(M,v) will denote that module M directly uses the variable v. *def(M,v)* will denote that module M directly defines the variable v. *trans(M,v)* will denote that module M is a transferring module of the variable v. Since a global variable may be defined by several modules and used by several modules, any definition of a global variable may be used by several using modules, and any use of a global variable may be defined by one of several defining modules.

The notation $A\text{->}B(p_i)$ represents that module A calls module B and passes an input parameter p to B, and $A\text{->}B(p_o)$ represents that module A calls module B and B returns an output parameter p to A.

The following analysis assumes:

(1) There exists an *oracle* for global variables. A *global variable oracle* of a module is a relation that specifies acceptable behavior of the global variable for any input to the module. It can be a functional representation, formal specification or correct version of the module, or a test analyst who knows the correct behavior of the global variable. Any uses of global variables should be included in the design specification, and their use should not be left to the discretion of programmers. This would prevent the common problem of the undisciplined usage of global variables. To identify the correct usage and behavior of global variables, a global variables oracle must be available.

(2) An incremental integration strategy will be used. As described in Appendix I, an incremental integration strategy is superior to a non-incremental strategy in that it is easier to locate faults where they occur.

(3) The test analyst has a reliable strategy for testing the correct usage of parameters. The following analysis focuses on the global variable testing problem and does not try to solve the parameter testing problem, which is beyond the scope of this thesis. Our objective is to transform the global variable testing problem to the parameter testing problem. Thus, if there exists a reliable testing strategy for parameters, the same strategy can also be applied to global variables testing.

Before presenting the strategy for testing global variables, we first present the basic testing problems for parameters. These testing problems can be classified into nine different cases, which can be reduced to a set of four basis cases.

7.2. Basis Cases for Testing Parameters

Parameter passing is one way that two modules can communicate with each other. The correct usage of parameters is tested during integration testing when they are actually being referenced in both the defining and using modules. The testing problem for parameters is complicated due to the presence of transferring modules. Some parameters that are passed to a module may not be directly referenced by the module; they are merely passed along to another module.

There are nine cases to be considered when testing a parameter passed between a calling module and its called module: the parameter is passed between

(1) a defining module and a using module,
(2) a defining module and a transferring module,
(3) a transferring module and a using module,
(4) two transferring modules,
(5) a using module and a transferring module,
(6) a transferring module and a defining module,
(7) a using module and a defining module,
(8) two defining modules, and
(9) two using modules.

The first four cases may be viewed as basis cases because each of the last five cases can be reduced to one of them. Case (5) can be treated as case (4) since the using module does not kill the definition and it can be viewed as a transferring module of the definition. Case (6) can be treated as case (4) if the defining module does not kill the definition; otherwise no testing is needed for this case. Case (7) can be reduced first to case (6) because the using module does not kill the definition, and then to case (4). Case (8) can be treated as case (2) if the called defining module does not kill the definition, or case (6) if the calling defining module does not kill the definition; otherwise, no testing is needed. Case (9) can be treated as case (3) since the calling using module does not kill the definition.

Of the four basis cases, only case (1) can be thoroughly tested during incremental integration testing, since the define and use of the parameter can be tested together. For case (4), it is impossible to test the parameter during either the unit or integration testing of the two modules because the parameter is not directly referenced. For both cases (2) and (3), the parameter should be extensively tested during unit testing of the defining module and the using module respectively; incremental integration testing is not effective in testing the parameter. Only after both the using and defining modules have been integrated can testing be performed on the parameter.

Because of the difference between input and output parameters, each basis case can be further divided into two subcases. Thus, there are a total of eight basis cases, which are listed below. Let A be the calling module which calls module B.

(1) $A \rightarrow B(p_i)$ & $def(A,p_i)$ & $use(B,p_i)$

The input parameter p is defined in module A and is used by module B. This is a common case that occurs in practice.

(2) $A \rightarrow B(p_i)$ & $def(A,p_i)$ & $trans(B,p_i)$

The input parameter p is defined in module A and will be used by some descendant of B; B is a transferring module of the definition of p in A.

(3) $A\text{->}B(p_i)$ & $trans(A,p_i)$ & $use(B,p_i)$

The input parameter p is used by module B; A is a transferring module of the definition of p from an ancestor of A.

(4) $A\text{->}B(p_i)$ & $trans(A,p_i)$ & $trans(B,p_i)$

Both modules A and B are transferring modules of the definition of p from an ancestor of A.

The next four cases are "mirror images" of the previous four cases with the output parameter being passed from the called module to its calling module.

(5) $A\text{->}B(p_o)$ & $def(B,p_o)$ & $use(A,p_o)$

This case is another common case one would expect to occur in practice; the output parameter is defined by module B and is used by module A.

(6) $A\text{->}B(p_o)$ & $def(B,p_o)$ & $trans(A,p_o)$

The output parameter is defined by module B and will be used by an ancestor of A; A is a transferring module of the definition of p in B.

(7) $A\text{->}B(p_o)$ & $trans(B,p_o)$ & $use(A,p_o)$

B is the transferring module of the definition of p from a descendant of B; A is the using module of p.

(8) $A\text{->}B(p_o)$ & $trans(B,p_o)$ & $trans(A,p_o)$

Both modules B and A are transferring modules of the definition of p from a descendant of B.

7.3. Global Variables as Parameters

This section shows that global variables can be treated as extra parameters for testing purposes. If the usage of global variables in various modules is explicitly documented, then the testing of global variables should not be more difficult than that for testing parameters. We first analyze the base cases involving only one define-use module pair, and then generalize the results to multiple define-use module pairs.

Figure 21 shows the different ways that a global variable may be referenced by two arbitrary modules. A rectangle represents a module and a circle denotes a global variable. An arrow from a module to another module indicates that the former calls the latter. An arrow from a module to a variable shows that the module directly defines the variable; the arrow is reversed for the case that the module directly uses the variable.

The relation between the defining module and the using module of the same global variable can be grouped into two major categories:

(1) Both the defining and using modules are on the same *chain of calls*. A *chain of calls* is a sequence of calls $Call(M_1)$, ..., $Call(M_k)$ such that module M_i may call module M_{i+1}, $1 \leq i \leq k-1$.

(2) The modules are on different chains of calls; in this case, they will have a *least common ancestor*. A *common ancestor* C of modules A and B is a module in the call graph which satisfies the following conditions:

- C and A are on the same chain of calls and C is called before A.
- C and B are on another chain of calls and C is called before B.

A *least common ancestor* L of modules A and B is a common ancestor of A and B such that no descendant of L is also a common ancestor of A and B.

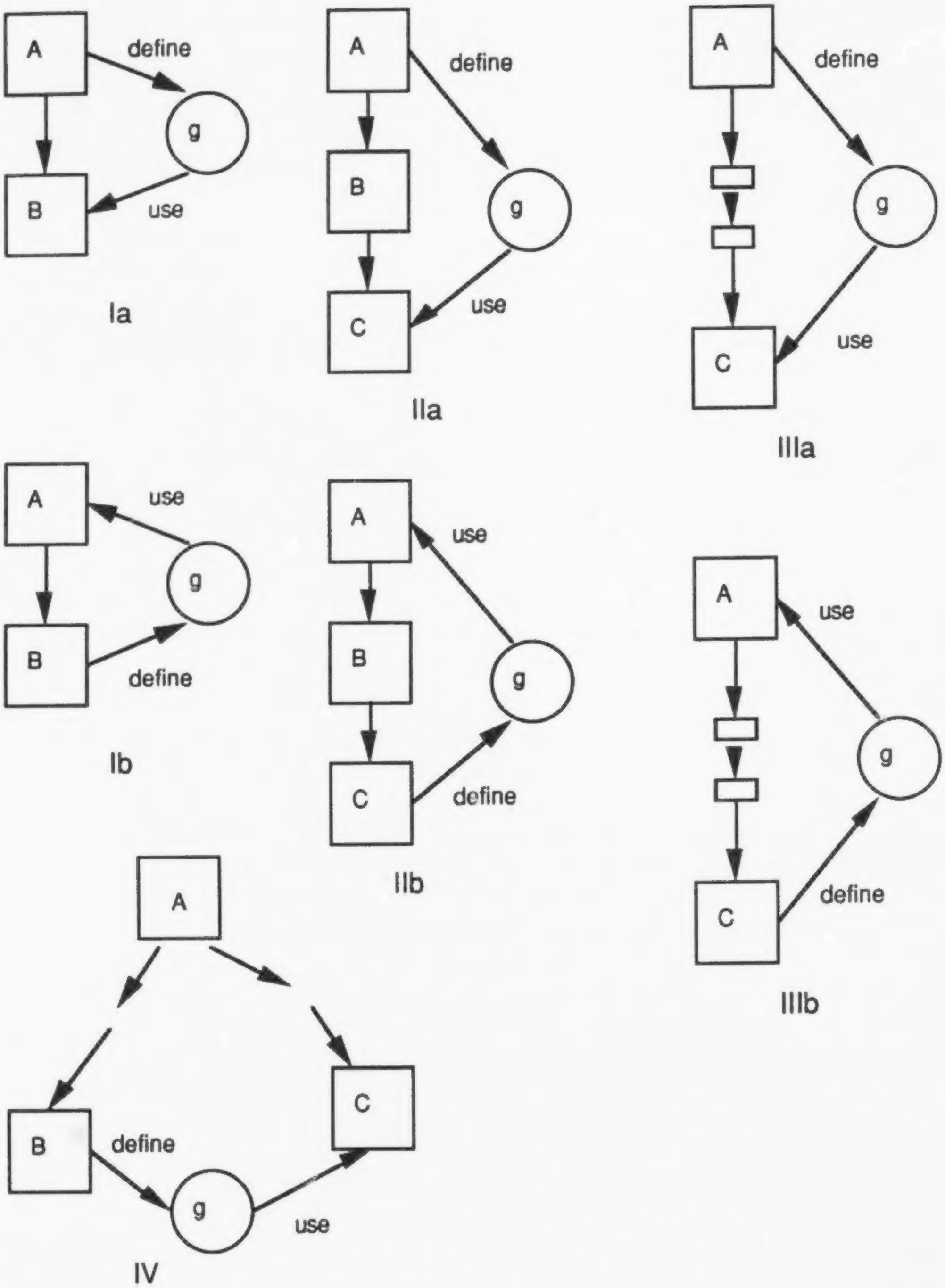

A
define
g
B
use
Ia
A
define
B
g
C
use
IIa
A
define
g
use
C
IIIa
A
use
g
B
define
Ib
A
use
B
g
C
define
IIb
A
use
g
C
define
IIIb
A
B
define
C
g
use
IV

Figure 21. Different Pairs of References to a Global Variable

For the sake of illustration, we have divided the first category into three classes: I, II, and III. In Class I, there is no module separating the define-use module pair. In Class II, there is exactly one module separating the define-use module pair. Class III represents the case of many modules separating the define-use module pair. Class IV shows the case when the defining module and the using module are on separate chains of calls.

Each of these classes can be shown to be a combination of the basis cases defined in Section 7.2. Thus, the existence of a global variable between a using module and a defining module can be viewed as an extra parameter being passed from the defining module to the using module via some transferring modules which may separate these two modules.

In the following analysis, we only consider the case when the definition of global variable g in each of the defining modules is executed before the use of g in the using modules in Figure 21. The reverse case can be ignored since no specific testing is required. Class Ia can be viewed as A passing the global variable g to B as an extra input parameter ($A\text{->}B(g_i)$ & $def(A,g_i)$ & $use(B,g_i)$). Class Ib can be treated as B returning an extra output parameter g to A, and can be represented by $A\text{->}B(g_o)$ & $def(B,g_o)$ & $use(A,g_o)$.

If we view the global variable as an input parameter of modules B and C, Class IIa can be viewed as a combination of basis cases (2) and (3), and can be represented by $A\text{->}B(g_i)$ & $trans(B,g_i)$ & $B\text{->}C(g_i)$ & $def(A,g_i)$ & $use(C,g_i)$. Similarly, Class IIb can be represented by $A\text{->}B(g_o)$ & $trans(B,g_o)$ & $B\text{->}C(g_o)$ & $def(C,g_o)$ & $use(A,g_o)$ and is a combination of basis cases (6) and (7), when the global variable is treated as an output parameter of modules B and C.

In Class III, there are several transferring modules between A and C. If we use E and F to represent any pair of consecutive modules on the chain of calls between modules A and C, the effect of these modules can be expressed by:

$trans(E,g_i)$ & $E\text{->}F(g_i)$ & $trans(F,g_i)$ when A is the defining module (case IIIa), and
$trans(E,g_o)$ & $E\text{->}F(g_o)$ & $trans(F,g_o)$ when A is the using module (case IIIb).

In Class IV, we only consider the case where module B is executed before module C because the case of module C executing before module B can be ignored since C will be using a definition of g from another module. In deriving the representation for Class IV, we have used E and F to denote any pair of consecutive modules on the chain of calls between modules A and B, and G and H to denote those between modules A and C. $trans(E,g_o)$ & $E\text{->}F(g_o)$ & $trans(F,g_o)$ is used to represent the passing of g among the transferring modules between B and A; $trans(G,g_i)$ & $G\text{->}H(g_i)$ & $trans(H,g_i)$ is used to represent the passing of g among the transferring modules between A and C. Table 5 summarizes the representation of the four classes in terms of the basis cases.

We next analyze the more complex situation when there are multiple define-use module pairs of the same global variable. We want to show that this situation is the same as a parameter being defined in many modules and used in many other modules. Consider the case of one defining module and multiple using modules. Each define-use module pair can be easily determined from an interprocedural data flow analysis [17, 25]. The previous analysis can be used to show that the global variable can be viewed as a parameter. Some define-use module pairs may not be feasible because the definition of the global variable may not reach the using module due to contradiction in the control flow condition.

Class	Representation	Basis Cases
Ia	A->B(g_i) & def(A,g_i) & use(B,g_i)	(1)
Ib	A->B(g_o) & def(B,g_o) & use(A,g_o)	(5)
IIa	A->B(g_i) & trans(B,g_i) & B->C(g_i) & def(A,g_i) & use(C,g_i)	(2) and (3)
IIb	A->B(g_o) & trans(B,g_o) & B->C(g_o) & def(C,g_o) & use(A,g_o)	(6) and (7)
IIIa	A->E(g_i) & trans(E,g_i) & E->F(g_i) & F->C(g_i) & trans(F,g_i) & def(A,g_i) & use(C,g_i)	(2), (3), and (4)
IIIb	A->E(g_o) & trans(E,g_o) & E->F(g_o) & F->C(g_o) & trans(F,g_o) & def(C,g_o) & use(A,g_i)	(6), (7), and (8)
IV	A->E(g_o) & trans(A,g_o) & trans(E,g_o) & E->F(g_o) & F->B(g_o) & trans(F,g_o) & A->G(g_i) & G->H(g_i) & trans(G,g_i) & H->C(g_i) & trans(H,g_i) & def(B,g_o) & use(C,g_i)	(6), (8), (4), and (3)

Table 5. Representing a Global Variable as a Parameter

For the case of multiple defining modules for one using module, we can easily generate all the possible define-use module pairs. Each such pair can be reduced to one of the four classes above. One complication with this case is that some definitions of the global variable may be killed by another defining module and some definitions may not reach the using module because of contradiction in the control flow condition. Interprocedural data flow analysis can be used to identify some of these occurrences.

Finally, for the case of multiple defining and multiple using modules, we can again identify all define-use module pairs. Each pair can then be reduced to one of the four classes considered earlier. In this case, the chance of a definition being killed is higher than the previous two cases and there will be more define-use module pairs. Nevertheless, the global variable can still be treated as an extra parameter being passed between the defining and using modules. From the results of the above analysis, a global variable can be treated as another parameter and tested accordingly.

7.4. Testing Global Variables

As described in Chapter One, the testing process is structured into three phases: unit testing, integration testing, and system testing. Global variables should be tested during unit and integration testing. We describe below some observations and a test selection strategy for testing global variables in each phase:

(1) Unit Testing

The global variable can be treated as an extra input/output variable to the module and the usual unit testing strategy can be used to test the global variable. Since the test analyst is assumed to have an oracle for global variables, he can judge the correctness of the test output with respect to the oracle. If the test analyst has a method for generating reliable tests for parameters, then he can use the same method for generating reliable tests for global variables. Let A be the defining module and B be the using module of a global variable g. In the case of module A, g is treated as an output variable and the test set should test every functional condition of A under which g might change; the test set would come from functional and structural tests which affect g. In the case of module B, the set of tests must range over sufficient values of g so that all effects on B are represented by these tests. Extremal values of g should be used whenever possible. To module B, g looks like another input variable.

(2) Integration Testing

As shown in Section 7.3, the global variable can be treated as a parameter and the normal integration testing strategy can be applied. For each pair of defining module A and using module B of the same global variable g, testing of g should be done after all the modules separating A and B have been integrated together. Functional tests may need to be *sensitized* from one module to another if A and B are on the same chain of calls, or to a least common ancestor if they are on different chains of calls. The test selection may be difficult if A and B are separated by many modules.

(3) System Testing

The existence of a global variable is not a concern here since system tests are based largely on the system specification and not on the implementation details of the software system. The use of a global variable will not complicate the testing strategy.

7.5. A Regression Testing Strategy for Global Variables

Many people have realized the importance of regression testing and recently several research studies in this area have been reported [23, 27, 47, 48, 79]. As in the case of most testing research, global variables are not treated in these studies. We present below a regression testing strategy for global variables:

(1) Unit Testing

For corrective regression testing, only those tests which traverse both the modified code and the instruction which references the global variable need to be repeated. Figure 22 shows the four possible relations between the reference to the global variable g and the changed code. Only cases 1 and 4 require regression testing of g. No new tests are required since the specification of the module is not modified. Observe that the specification states the correct usage of the global variable and since the specification is unchanged, the previous tests for the global variable should be valid.

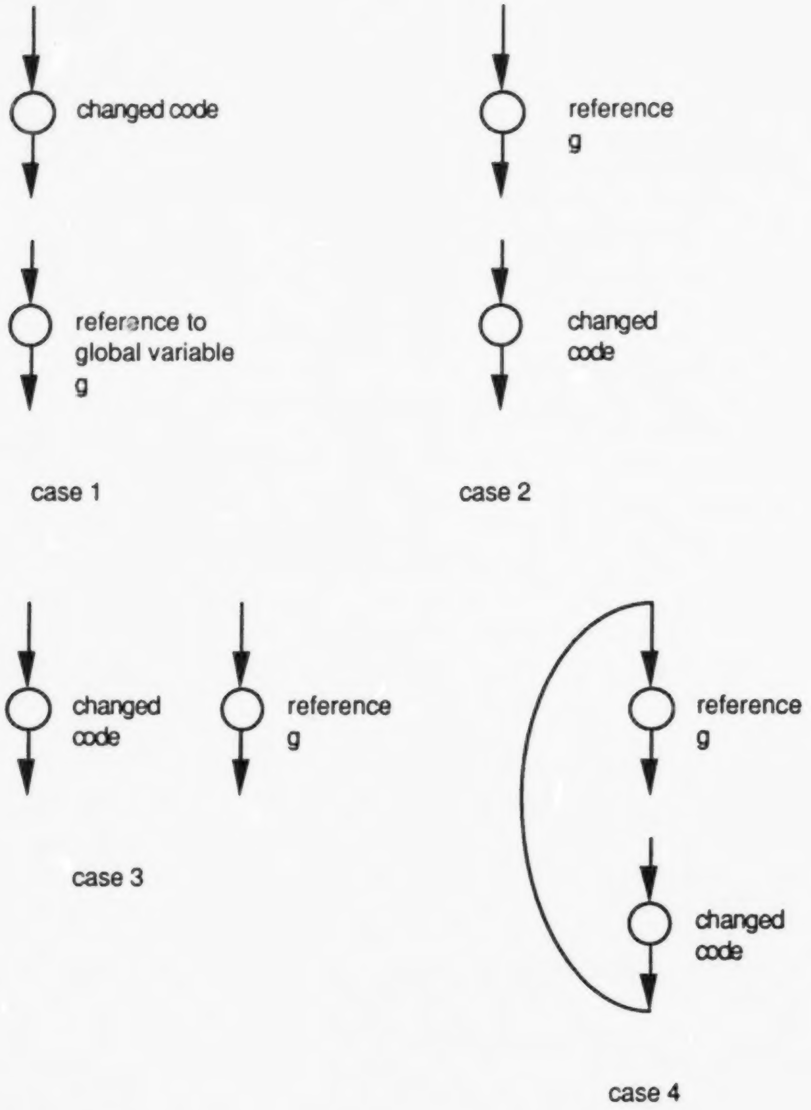

changed code
reference to global variable g
case 1
reference g
changed code
case 2
changed code
reference g
case 3
reference g
changed code
case 4

Figure 22. Relations Between a Global Variable and Changed Code

Let A be the defining module and B be the using module of a global variable g. From the previous discussions, if just the implementation of A or B changes, but the respective specification does not change, all we need to do is to rerun the given global variable tests for A or B, respectively. In the case of module A, we should repeat the global variable tests and check whether the previous values of g are replicated. In the case of module B, we run the given global variable tests on g to check if the output of B indicates any error. In both cases, if the tests give different results, then either the specification should be modified or the changes are incorrect.

For progressive regression testing, if the specification change involves the global variable, then new tests should be created; otherwise the retesting strategy is the same as in corrective regression testing.

(2) Integration Testing

If a global variable g is used between modules which are modified, we need to apply the integration techniques described in Chapter Five, using g as an input or output at the appropriate point in the integration testing. The testing of g should be re-applied if the modified module references g; otherwise, no global variable testing is required.

If a defining module of g is modified, then the change may affect the definition of g which is used by other using modules. In this case, integration testing should also be performed with each of the using modules of g. If a using module of g is modified, then the change may affect the way that g is used. In this case, integration testing should be performed with each of the defining modules of g to check whether the corresponding definition and use assumption has not been violated. Some examples are shown in Figure 23.

If the modified module has undergone code modification only, then some previous tests can be repeated and the correctness of the global variable usage can then be checked. If the specification is changed, new tests may be needed. It may be difficult to design new tests but it is no different from the initial testing problem.

(3) System Testing

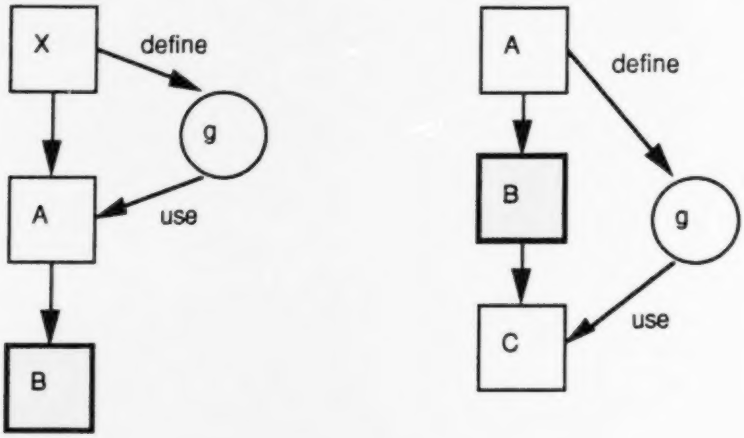
As explained in Section 7.4, global variables can be ignored in this testing phase.

This chapter has shown that global variables can be treated like parameters for testing purposes and can be tested accordingly. Insights into testing and regression testing global variables have been described. An important assumption of our strategies is the requirement that the global variable usage be explicitly stated in the design specification. The adoption of this requirement entails two major advantages:

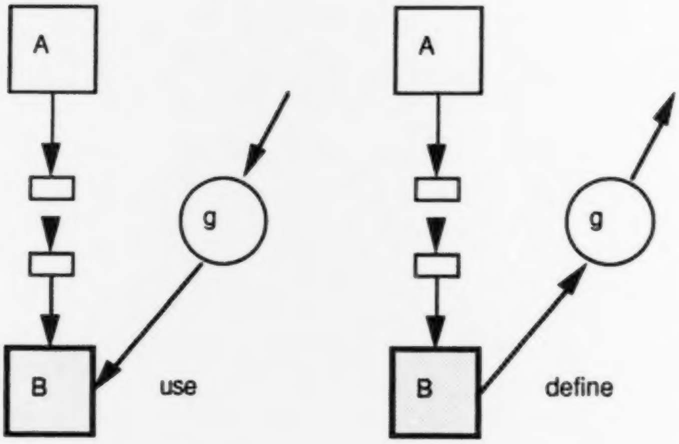
(1) The usage of global variables is controlled. The programmer is not allowed to include unplanned and undocumented use of global variables. The interactions of global variables and other components of the program are no longer hidden in the code.

(2) Since the effect of global variables is known, there is no excuse for not testing them. A common excuse for not testing global variables is that they may affect many parts of the program and since there are many possible areas for testing their effect, it is impossible to test them properly.

If one accepts the fact that an unknown program property cannot be thoroughly tested, then the effects of global variables must be known for testing purposes. Once the effects of global variables are known, testing them is not more difficult than testing parameters.


X
define
g
A
use
B
A
define
B
g
C
use

Global variable testing is not needed
(In all cases, module B is the only one changed)


A
g
B
use
A
g
B
define

Global variable testing is needed in this two cases

Figure 23. Some Examples Where Global Variables Testing are Required

Chapter Eight

A Regression Testing System and a Cost Model

Because of the high volume of testing necessary to insure reliability, a test analyst cannot rely solely on any manual testing approach; some form of automated system is needed to ease the burden of testing. An automated system eliminates the labor intensive and error-prone manual approach. Also, as described in previous chapters, regression testing involves retaining tests and reusing some of them after each modification to the software. This process involves many book-keeping chores which are especially suitable for automation.

A large number of testing tools exist to assist in software development [66]; however, very few of them can be applied directly to regression testing. Among those which claim to be useful for regression testing, most provide no more than the capability to store the previous tests and rerun them after every modification [12, 60, 69, 70]. They do not provide any intelligent test selection capability, nor any estimation of the required testing effort. Recently, several tools were introduced for regression testing at the unit level [3, 23, 79]. They do not automate the entire regression testing process.

This chapter presents the design of a regression testing environment which addresses all phases of regression testing and supports both black-box (functional) and white-box (structural) testing. Section 8.1 presents the design objectives and an overview of our regression testing system - ReTestS. Section 8.2 details each component of ReTestS.

This chapter also analyzes the relative cost of different regression testing strategies. It is not obvious that a *selective* regression strategy will always require less effort than the retest-all strategy. A *selective* regression testing strategy selects and repeats a subset of the previous tests, rather than repeating all previous tests. Our regression strategy is an example of a selective regression strategy. Section 8.3 provides a model of the cost of testing and compares the relative cost of the retest-all and the selective regression testing strategies. The conditions under which one strategy is more economical than the other strategy are established.

8.1. An Overview of ReTestS

This section provides a high level view of our **Re**gression **Test**ing **S**ystem, *ReTestS,* which is designed with the following objectives:

(1) It should support a full spectrum of services required for regression testing, ranging from unit to system testing, and including both specification-based and structural-based testing. A modified program not only requires regression unit testing, but also regression integration and regression system testing. A "complete" regression tool should assist in all three testing activities. Also, since a single testing technique is not effective in testing a program, a regression

tool should support both specification-based and structural-based testing techniques.

(2) It should provide an interactive, easy to use, well-integrated and automated environment with information (for example, tests and program analysis) shared among the various tools.

ReTestS acts as a regression testing assistant which handles most of the routine chores of regression testing, such as static analysis, test execution, and update of the test plan. We envision that ReTestS will be used according to the following scenario: After the modifications are entered into ReTestS, ReTestS performs a static analysis of the new version of the software, comparing it to its previous version. The sets of reusable, retestable, and obsolete tests for all three testing phases are then determined based on the stored test plan and change information. ReTestS automatically executes the retestable tests, evaluates the test coverage, identifies the program components that remain to be covered, queries the test analyst for new tests if the testing criterion is not satisfied, collects the testing results, and updates the test plan with the new dynamic information. Using ReTestS, the test analyst's major task is to create new tests, as this cannot be completely automated.

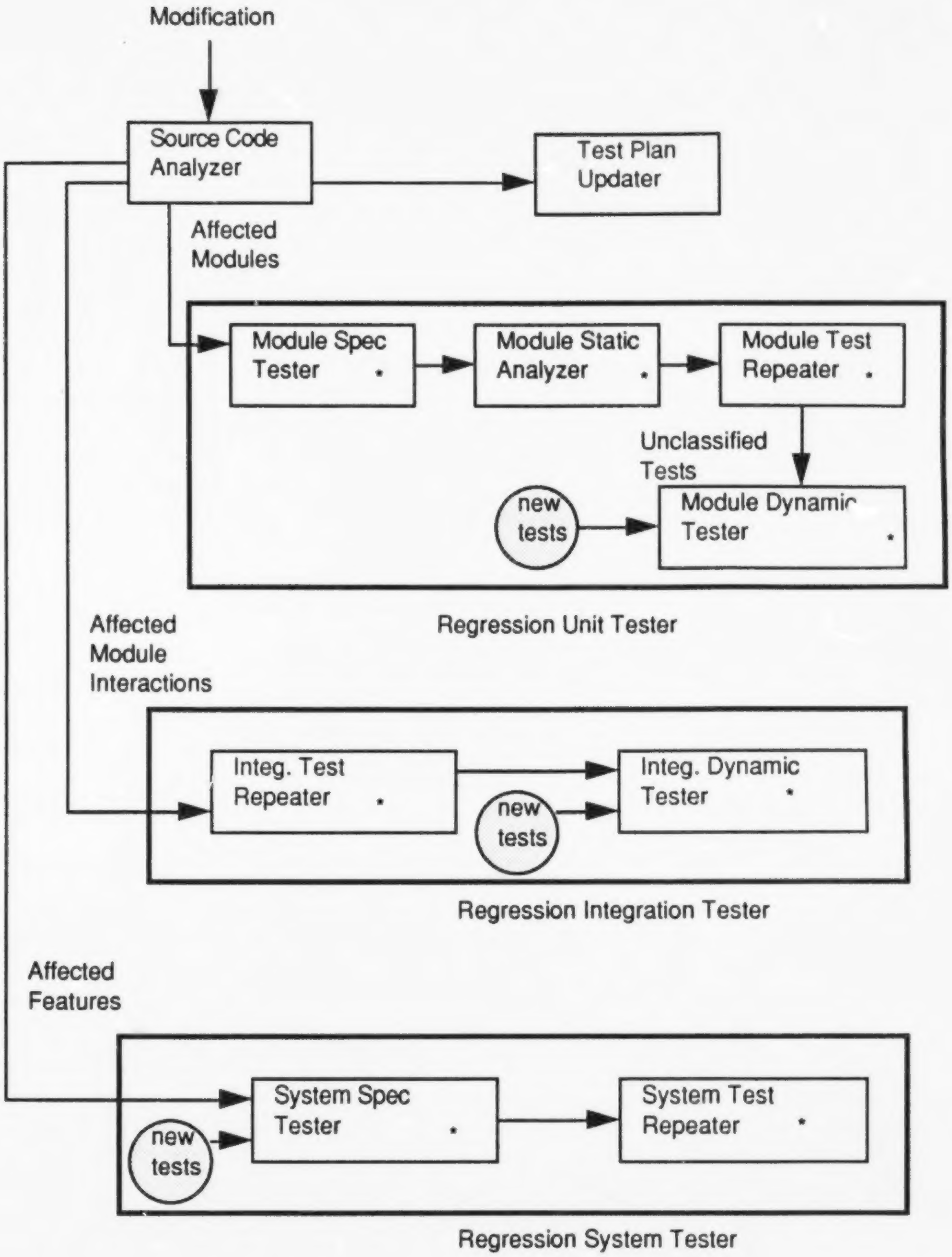
The structure of ReTestS is depicted in Figure 24. ReTestS consists of three separate regression testers, one for unit, integration, and system testing, respectively. The unit regression tester implements the strategy outlined in Chapter Four and consists of the Module Spec Tester, Module Static Analyzer, Module Test Repeater and Module Dynamic Tester. The regression integration tester implements the strategy described in Chapter Five and consists of the Integration Test Repeater and Integration Dynamic Tester. Finally, the regression system tester implements the strategy outlined in Chapter Six and consists of System Spec Tester and System Test Repeater.

With separate components for each phase of testing, the modular design of ReTestS allows each tester to be implemented independent of the others and permits independent upgrading of any single tester when new technology is available.

The main control module of ReTestS is the Source Code Analyzer whose major functions include:

- interacts with the test analyst
- collects the changes information
- identifies affected modules, affected module interactions, affected features, and invokes the regression unit tester, regression integration tester, and regression system tester.

Another unique module, the Test Plan Updater, provides bookkeeping services to all other modules. Other modules invoke the Test Plan Updater whenever data such as tests, program structure, and test execution history need to be updated.


Modification
Source Code Analyzer
Test Plan Updater
Affected Modules
Module Spec Tester *
Module Static Analyzer *
Module Test Repeater *
Unclassified Tests
new tests
Module Dynamic Tester *
Regression Unit Tester
Affected Module Interactions
Integ. Test Repeater *
Integ. Dynamic Tester *
new tests
Regression Integration Tester
Affected Features
System Spec Tester *
System Test Repeater *
new tests
Regression System Tester

* Invoke Test Plan Updater

Figure 24. Structure of ReTestS

8.2. Components of ReTestS

The components of ReTestS can be classified into three categories:

(1) Analyzers (for example, Source Code Analyzer and Module Static Analyzer) do not execute the program, but carry out an analysis of the program and changes based on the stored information. In particular, they update the data flow and control flow relations of affected modules, and the calling structure of the system. They identify the specific changes and classify the tests.

(2) Test Repeaters (for example, Module Test Repeater, Integration Test Repeater, and System Test Repeater) are unique to regression testing. They first select the tests using the reuse selection criterion, which is based on the all-essential assumptions. They then execute the program with the retestable tests, and update the dynamic behavior of the program. No interaction with the test analyst is needed. Test Repeaters, unlike most regression testing tools, only repeat the tests that will exercise the modification and do not always repeat all previous tests.

(3) Testers (for example, Module Spec Tester, Module Dynamic Tester, Integration Dynamic Tester, and System Spec Tester) are concerned with new test construction. They query the user for new tests, execute them, and store the tests and the dynamic behavior of the program. Observe that the functions of the Testers are exactly those required during the testing phase. It is expected that some existing testing tools can be easily modified as Testers for ReTestS.

We next give an overview of each component of ReTestS.

Source Code Analyzer

Purpose: To identify the affected modules, affected module interactions, and affected software features, and update program information.

Input: Program modification and the affected software features.

Output: Affected modules, affected module interactions, and affected software features.

This component retrieves the program structure and feature information from the database, and analyzes and compares the new source code to the existing code. Its main function is to identify the affected modules, affected module interactions, and affected software features. It iteratively invokes Module Spec Tester, passing down one affected module at a time. After regression unit testing is completed, the Source Code Analyzer then passes down one affected module interaction at a time to the Integration Test Repeater. Finally, on completion of the regression integration testing, it invokes the System Spec Tester, once for each affected software feature.

Module Static Analyzer, Module Test Repeater and **Module Dynamic Tester** are described in Section 4.1.

Module Spec Tester

Purpose: To test the modified specification of an affected module.

Input: Modified subspecs, deleted subspecs and new subspecs of an affected module, and new tests.

The algorithm for the Module Spec Tester is given in Section 4.2.1. This component is invoked if the design specification of the affected module is modified. It queries the test analyst for new specification tests and executes them. Its operation is very much like the Module Dynamic Tester, but it only works with specification-based tests.

Integration Test Repeater

Purpose: To repeat the retestable tests.
Input: An affected module interaction.
This component identifies the basis case which corresponds to the affected module interaction. It then sets up the integration testing environment by retrieving the previous drivers and stubs, if required, for testing the module interaction. One function of this component is to update the stored testing environment. All tests which are usable at level 1 or 2 are identified and repeated.

Integration Dynamic Tester

Purpose: To complete testing of the affected module interaction.
Input: An affected module interaction and new tests.
This component queries the test analyst for additional tests for testing the affected module interactions. It also updates the integration testing environment. Another function of this module is to assist the test analyst in sensitizing the tests by performing symbolic evaluation of program paths.

System Spec Tester

Purpose: To test the affected software feature.
Input: The affected software feature and new system tests.
This component queries the test analyst for new system tests which can be used to validate the software specification changes or to satisfy the structural-interface criterion. This tester will be invoked if the specification is modified or the structural-interface criterion is not satisfied.

System Test Repeater

Purpose: To repeat some existing tests for checking the behavior of the system.
Input: The affected modules.
This component automatically identifies and repeats the retestable tests. The retestable tests will depend on the chosen structural-interface criterion and the software modification.

Test Plan Updater

Purpose: To update all the information needed for regression testing.
This component is called by other components to update stored information about the program, tests, and testing environment.

ReTestS addresses the full spectrum of tasks in a typical regression testing process, and provides many automated tools to minimize the regression testing effort. Although it provides much support specifically for regression testing, ReTestS may also be used for testing since several components of ReTestS deal with new tests and the testing process. We summarize below the services provided by ReTestS:

(1) Identify required initialization
ReTestS can identify the parameters and global variables which require initial values. These are accomplished by data flow analysis [60]. It can also identify the required drivers and stubs to be supplied by the test analyst. The input to ReTestS includes: change information, any required initial values for parameters and global variables, new drivers and stubs, and new tests.

(2) Select test
ReTestS automatically identifies the reusable, retestable and obsolete tests using the reuse selection criterion.

(3) Execute test, analyze and update results
ReTestS performs test execution of retestable tests and new tests entered by the test analyst, compares the output with the expected results, and collects and updates all program information, testing environments, and test results.

(4) Compute code coverage
ReTestS computes the structural coverage at the unit level and the structural-interface coverage at the system level, and informs the testing analyst when the new selection criterion is satisfied, so that testing can be terminated.

(5) Assist in test design
ReTestS assists in test design by informing the test analyst which program components remain to be executed and by performing symbolic evaluation to help sensitize tests.

The construction of ReTestS is a worthwhile project despite the fact that it is complex and time-consuming. Some components of ReTestS may be created from existing testing tools with minor extensions. For example, the Module Dynamic Tester can be developed by modifying either the data flow tester of Korel and Laski [42] or the incremental data flow tester of Harrold and Soffa [24].

Other components of ReTestS may make use of other tools described in the literature. The symbolic execution tool of Clarke and Richardson [8] can be used for sensitizing the test input. The Source Code Analyzer may be built on top of Kuhn's [43] source code analyzer for computing both the calling relation of the system and the data flow and control flow relations within each module. The system introduced by Dogsa and Rozman [12] may be used to generate the driver module code.

Unfortunately, the components for implementing integration and system testing are new and will need extensive development effort. We estimate that the development time for ReTestS to be at least two man-years.

8.3. Comparing the Relative Cost of Regression Testing Strategies

A practical question that has not been answered is the cost of applying a particular regression testing strategy. It seems that a selective strategy will require more time and resources for test selection in order to realize a reduction in the number of tests executed. A benefit is accrued only if the effort spent in test selection is less than the cost for executing the extra tests. This section provides an analysis of the trade-off between using a selective strategy and the retest-all strategy.

The cost of applying any testing strategy will be modeled in Section 8.3.1. Section 8.3.2 compares the relative cost of a selective strategy and the retest-all strategy. The conditions under which the selective regression testing strategy is less costly than the retest-all strategy will be derived.

8.3.1. A Test Cost Model

Testing requires not only test selection, test execution and result analysis, but also understanding the software system in order to determine the system behavior to be tested. The cost of applying a set of tests to a software system consists of the following components:

(1) *System Analysis Cost, Ca*
Before a test set can be selected, the test analyst must become familiar with the system specification, design and possibly the program. Time must be spent studying the various requirements and design documents. The knowledge gained in this phase will also allow the test analyst to judge whether the program behavior is correct. The larger the system under test, the higher is this cost.

(2) *Test Selection Cost, Cs*
After becoming familiar with the expected behavior of the system, the test analyst can then select the tests for testing the actual system behavior. Some cost is incurred in working out the test input, and identifying the correct output or system behavior. This cost component will depend largely on the chosen test strategy.

(3) *Test Execution Cost, Ce*
Test execution cost includes the cost of setting up the environment for testing (such as loading and compiling required modules and entering the proper data tables) and the cost of computing resources for the actual execution of the system under test. The cost can be quite high for some applications. For example, in the telecommunication industry, the cost of setting up a testing lab to simulate an actual communication network can be as high as several million dollars.

(4) *Result Analysis Cost, Cr*
The last step of testing involves checking the behavior of the system under test against the specified behavior. The cost factors for Cr are: the test analyst's time in collecting the test outputs, the time to compare the test output to the system specification, and the computing resource required for recording the system behavior.

These cost components are dependent on several factors and some of them are interrelated. The testing strategy used has a direct effect on the cost. For example, since a black-box strategy only requires an analysis of the specification, the system analysis cost for such a strategy will be less than that of a testing strategy which uses both white-box and black-box testing. Note that the number of tests also depends on the test strategy. A test strategy which requires that every definition-use pair be executed generally needs more tests than one which only requires all instructions be executed.

All test cost components are related to the number of tests. Although it seems that Ca is independent of the number of tests because the cost of understanding both the problem specification and the system is roughly the same for designing either 10 tests or 100 tests, one can argue that a complex system will require a high cost for understanding and also many more tests than a simple system. Thus, the number of tests and the system analysis cost are related to the same factor - the complexity of the system.

Also, the other three cost components, Cs, Ce, and Cr, should be dependent on the number of tests; the higher the number of tests, the higher are the respective costs.

The system analysis and the result analysis costs are related in a subtle way. Suppose the test analyst has spent a small amount of time in the system analysis phase. When he does the result analysis, he will have to spend more time to determine whether the output is correct because he had to first understand the expected program behavior by studying the specification and other program documents. Thus by reducing Ca, it is likely Cr will increase. By doing a thorough job at the system analysis phase, the test analyst will know the expected behavior of the system and it will be straightforward to check whether the program output is correct during the result analysis phase.

From the previous discussion, we will model the result analysis cost Cr as consisting of two sub-components: the cost Cu for understanding the program and specification in order to judge whether the program behavior or output is correct, and the cost Cc for comparing the test output to the expected output. The *checking cost Cc* should be proportional to the number of tests. Thus, Cr = Cu+Cc. Observe that Cu is directly related to Ca because a thorough system analysis upfront will reduce the effort for understanding the system during the result analysis phase.

In summary, our model is based on two key assumptions:

(1) the cost of applying a test strategy *ts* which uses a set T of tests is

$$C(ts) = Ca(T)+Cs(T)+Ce(T)+Cu(T)+Cc(T)$$

(2) Cs, Ce and Cc are linearly dependent on the number of tests.

8.3.2. Cost of Regression Strategies

In this section, we first analyze the cost of applying the retest-all strategy and then the cost of the selective regression strategy. The retest-all strategy uses two test sets: all old tests (To) and a set of new tests (Tn) for the modified specification, or modified code, or both. Thus the testing cost is

$$\begin{aligned} C(\text{retest-all}) = {} & Ca(To)+Cs(To)+Ce(To)+Cu(To)+Cc(To) \\ & +[Ca(Tn)+Cs(Tn)+Ce(Tn)+Cu(Tn)+Cc(Tn)] \end{aligned}$$

Since according to the retest-all strategy, the test analyst does not analyze the previous tests before applying them and no effort is needed to select these tests, Ca(To) = Cs(To) = 0. However, as argued in Section 8.3.1, one consequence of skipping the system analysis and test selection phases is that the test analyst will pay a heavy price later in the result analysis phase; that is, Cu(To) will be high. Thus the cost of applying retest-all can be reduced to

$$C(retest\text{-}all) = Ce(To)+Cu(To)+Cc(To) \\ +[Ca(Tn)+Cs(Tn)+Ce(Tn)+Cu(Tn)+Cc(Tn)]$$

Next, consider the cost of applying a selective regression strategy where a subset Ts of To, together with the new tests Tn, are used to test the modified software system.

$$C(selective) = Ca(Ts)+Cs(Ts)+Ce(Ts)+Cu(Ts)+Cc(Ts) \\ +[Ca(Tn)+Cs(Tn)+Ce(Tn)+Cu(Tn)+Cc(Tn)]$$

We will assume that the sum of Ca(Ts) and Cu(Ts) is approximately the same as Cu(To), because in both cases the test analyst has to understand the effect of the modification on the validity of the previous tests. Also notice that the second set of terms in C(retest-all) and C(selective) are identical.

Based on these observations, C(selective) is less than C(retest-all) if

$$C(selective) - C(retest\text{-}all) < 0$$

or

$$Cs(Ts)+Ce(Ts)+Cc(Ts)-[Ce(To)+Cc(To)] < 0 \qquad (1)$$

because $Ca(Ts)+Cu(Ts) \cong Cu(To)$. (1) can be rewritten as

$$Cs(Ts) < [Ce(To)-Ce(Ts)]+[Cc(To)-Cc(Ts)] \qquad (2)$$

In other words, the selective strategy is more economical than the retest-all strategy if the cost for selecting a subset of the previous tests is less than the cost for executing and checking the extra tests needed for the retest-all strategy.

To gain more insight into inequality (2), we will invoke the second assumption: Cs(T), Ce(T) and Cc(T) are directly proportional to the number of tests in test set T. Thus, we write

$$Cs(T) = s\,|T| \\ Ce(T) = e\,|T| \\ Cc(T) = c\,|T|$$

where s, e and c are constants representing the cost per test for the selection cost, execution cost and checking cost, respectively. The exact values of these constants will depend on the complexity of the system, the particular implementation, and the testing strategy. Also, these constants may vary with the experience and ability of the test analyst.

Substituting the above expressions, we can rewrite inequality (2) as

$$s\,|Ts| < e\,(|To| - |Ts|) + c\,(|To| - |Ts|) \\ = (e+c)\,(|To| - |Ts|)$$

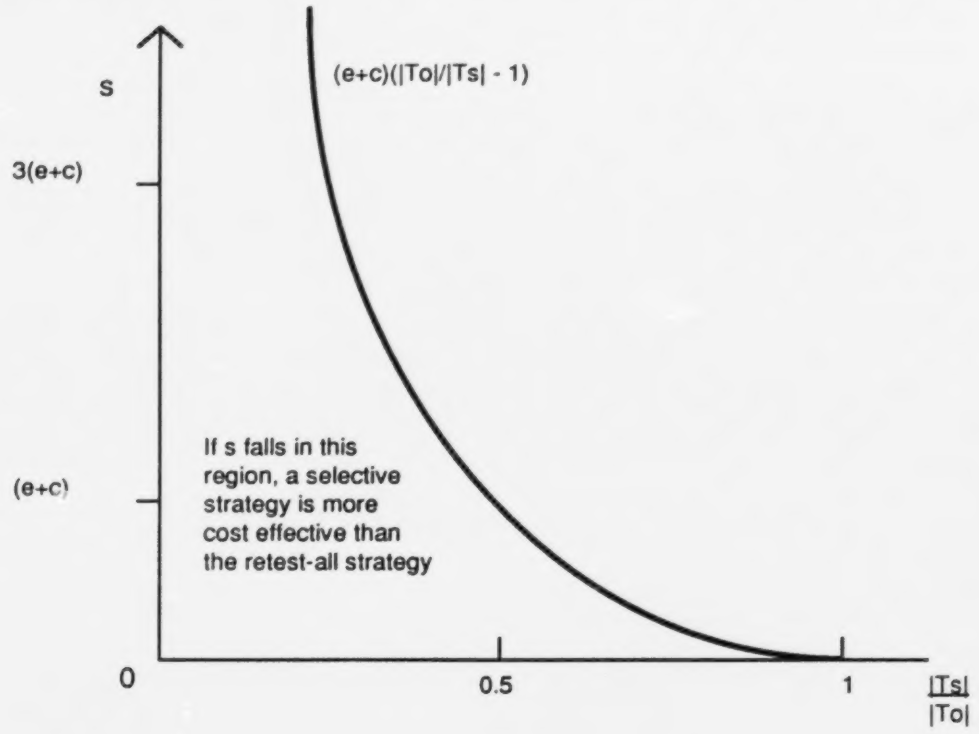
or

$$s < (e+c)\ (|To| / |Ts| - 1) \qquad (3)$$

The inequality (3) depends on the values of s, e, c and the ratio of $|Ts|$ to $|To|$. Suppose the selective strategy only repeats half of the previous tests ($|Ts| = 0.5\ |To|$), then it is more cost effective than the retest-all strategy if

$$s < (e+c)$$

Figure 25 summarizes the conditions under which the selective strategy is more economical than the retest-all strategy. The vertical axis represents s values and the horizontal axis denotes the ratio $|Ts| / |To|$. The graph $(e+c)(|To| / |Ts| -1)$ is plotted against $|Ts|/|To|$. If the selection cost of a selective strategy falls below the graph, using such a strategy will provide some saving in test cost over the retest-all strategy. For example, if $s = (e+c)$ and the selected $Ts = 0.4\ To$, then a selective regression strategy will cost less than the retest-all strategy.



If s falls in this region, a selective strategy is more cost effective than the retest-all strategy

Figure 25. Cost Relationship Between the Retest-all and Selective Strategies

For a given change situation, inequality (3) can be used to select the more cost effective regression testing strategy. The values for e, c and s can be determined according to the project type from historical data. Since |To| is known from previous testing, the only parameter to be determined is |Ts|. |Ts| can be estimated by an analysis of the extent of changes to the system specification, requirement, and source code.

Chapter Nine

Empirical Results

A first step in evaluating a new test strategy is to establish its effectiveness in detecting errors. It would be desirable to apply our strategy in a realistic situation to assess its practicality. Unfortunately, any experimentation could be misleading without a large maintenance database. The retesting data and results collected regarding the maintenance activity of a particular program may not be representative of the maintenance activity in other programs. The major problem is that there are numerous types of maintenance tasks and a reliable validation must include experimentation on a significant number of various types of programs and a significant number of different modifications. A single experiment will not be representative of a broad class of maintenance activities. Given this reality, many proposed solutions to the various maintenance problems are accepted largely based on intuitive arguments, relying on programming practices and some general assumptions about the change and testing process [78].

Section 9.1 presents a case study using our regression testing strategies for unit, integration and system testing. Although the results may not be conclusive, as discussed earlier, it is better to have some confirmation of the applicability of our concepts than none at all. This study aims to investigate the effectiveness of our regression testing framework, and the amount of saving in test effort, in terms of the number of tests, over the traditional retest-all strategy.

Section 9.2 describes data collected from a telecommunication company to validate the linear assumptions of the test cost model. The data suggests that the test selection cost, execution cost and result checking cost may be treated as linearly dependent on the number of tests used.

9.1. Applying the Regression Strategy

Section 9.1.1 describes the program used in our experiment. Sections 9.1.2 and 9.1.3 discuss the experiment and the testing of this program. A description of the four modifications and the regression testing experience are outlined in Section 9.1.4. The results of the experiment are summarized in Section 9.1.5

9.1.1. The Program Under Test

The program under test is a database program which provides five major functions. The user can:

- create a student database which records the student names, identification numbers, and assignment mark,
- enter new student records and new student marks,

- update a single assignment mark or all the marks of the students,
- delete a student record, and
- display statistical information about the class marks, including the highest, lowest, and average marks for each assignment, the average mark of each student, and the overall average marks of all the assignments.

The program also checks for valid student identification numbers and assignment numbers.

The student database is stored in two files. A *Headerfile* stores the number of student and the number of assignment entered. A second file, the *Studentfile,* stores all the student records in the form of *Student Name, Student Id, mark 1, mark 2, ..., mark n,* where n represents the maximum number of assignments and is controlled by the constant *MaxAssign*. The maximum number of student records is controlled by the constant *Max*. The student id is constrained by the constants minimum identification number (*MinId*) and maximum identification number (*MaxId*).

The original version of this interactive program has been developed and used by an instructor, an experienced programmer analyst, to demonstrate some database concepts. The actual version used for our experiment contains a few minor modifications.

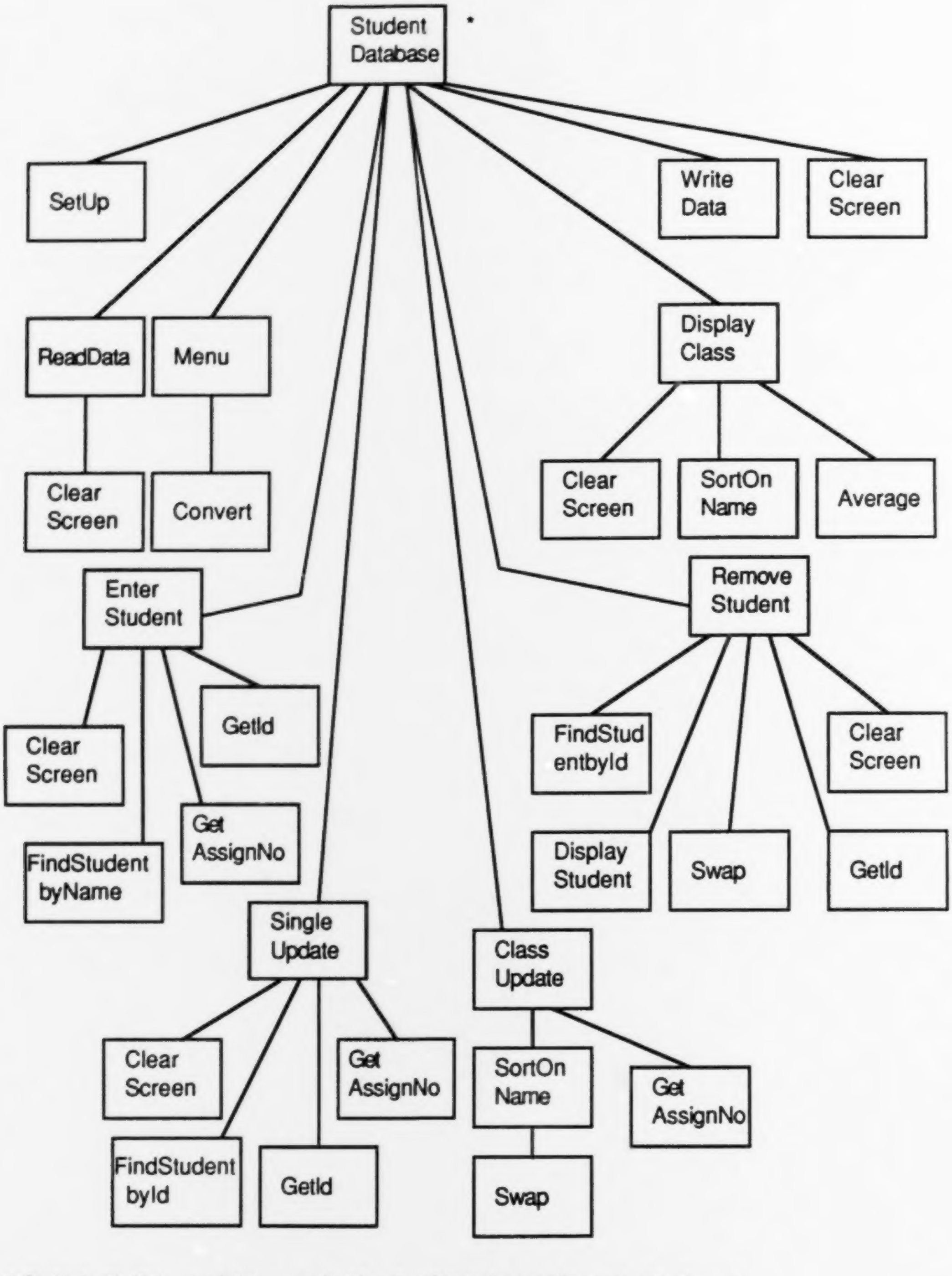
Figure 26 gives the call graph of program *StudentDatabase*. This program consists of twenty distinct modules, thirty-one module interactions which involve the use of thirty-two modules and over 550 lines of Pascal code. The calling order graph of program *StudentDatabase* is given in Figure 17 of Chapter Six and a subsection of the input graph is shown in Figure 13. This program contains 6 software features, 13 simple call paths, and 48 simple input paths.

This program is chosen for our experimentation for the following reasons:

- Its size is manageable. Since we have not implemented ReTestS, most of the testing and regression testing are performed mainly by hand. A large program will require a long time to analyze and test.
- The program is realistic, and its application is understood by most readers, thus reducing the effort in comprehending our results by other researchers.
- Since the user is interested in improving the program and is available for additional consultation, it is easy to obtain feedback from the actual user. In fact, most of the modifications used for our study are enhancements suggested by the user, and not just arbitrary modifications.

9.1.2. The Experiment

The experiment was carried out manually for the data flow static analysis, insertion of probes, and result analysis. First each module was analyzed, and the all-use pairs were identified and stored in a table. We then instrumented the program by hand to collect the structural coverage information. The monitor statements were print statements which wrote the branch identifiers to a trace file. The instrumented program was then executed with the selected tests. The trajectory information was written to a file for later analysis. During the initial testing, we created various tables for storing the functional tests of each subspec of the module, the feature-specification mapping, the feature-module matrix, the feature-test matrix, and the module-test matrix. These tables were kept up to date after each modification. Most modules have only a few subspecs.


Student Database
*
SetUp
Write Data
Clear Screen
ReadData
Menu
Display Class
Clear Screen
Convert
Clear Screen
SortOn Name
Average
Enter Student
Remove Student
GetId
Clear Screen
FindStudentbyId
Clear Screen
FindStudent byName
Get AssignNo
Display Student
Swap
GetId
Single Update
Class Update
Clear Screen
Get AssignNo
SortOn Name
Get AssignNo
FindStudent byId
GetId
Swap

* Calls to Built-in modules supplied by the Pascal complier are not shown

Figure 26. Call Graph of Program StudentDatabase

After each modification, the changed program was analyzed, all-use pairs were updated based on the changed code, and new monitors were inserted into the new version to collect trajectory information about the new instructions. The previously stored tables were analyzed to identify the reusable tests. New tests and retestable tests were then run and new trajectory information was collected. New tests were added until the testing criteria (all-use criterion for unit testing, and all-simple-input-paths-and-branches and all-simple-call-paths-and-branches criteria for system testing) were satisfied.

Different types of monitors were used for unit testing and system testing, since the former involved the tracking of define-use relations while the latter needed to track the calling relations. Thus, during unit testing, the program was instrumented with monitors which allowed us to deduce the all-uses coverage. For system testing, the program was instrumented with a different set of monitors to track mainly the execution order of the modules and the input instructions. The system level monitors were print statements which wrote the module name or the input instruction identifiers to a trace file. The program was not instrumented for integration testing.

9.1.3. Testing the Original Program

During unit testing, functional tests were first created from the specification of each module. Additional structural tests for satisfying the *all-uses* structural coverage criterion were then created [61]. Informally, a test set satisfies the all-uses criterion if the instruction trajectories of these tests include at least one definition-clear subpath from each definition to each use reached by that definition and each successor node of that use. The requirement of traversing the successor nodes is to force all branches to be taken following a predicate. Weyuker [74] has shown that, in the worst case, the all-uses criterion requires at most a number of tests which is quadratic in the number of conditional instructions of the module being tested. However, a recent study has shown that in practice the number of tests is linearly dependent on the number of conditional instructions [75]. Our results also indicated that only a small number of tests were needed to satisfy the all-uses criterion.

Most of the unit tests (approximately 90%) were functional tests. In most cases, we only needed a few additional structural tests to satisfy the structural criterion. In designing the unit tests, extremal values of the input variables were used for all the functional tests whenever possible. This helped to reduce the number of required tests since each test can be testing a function and a boundary of an input domain simultaneously. Besides using this simple strategy to reduce the number of tests, no other minimization of the number of tests was performed.

During integration testing, we applied both functional and extremal tests, as described in Chapter Five. A bottom-up integration strategy was used. In most cases, the integration tests were just the union of the unit tests of the two modules involved in the module interaction.

System tests were designed mainly to test all the software features, and at the same time, were required to satisfy the structural-interface criterion. The all-simple-input-paths-and-branches and the all-simple-call-paths-and-branches structural-interface criteria were used. This SI criterion was found to be easily satisfied by the functional tests. Approximately 10% of the system tests were selected specifically to cover the components required by the SI criterion.

No special testing for global variables was needed because all global variables were either program constants or were initialized (defined) once in the program. The normal unit and integration testing automatically tested them. The total number of unit tests, integration tests, and system tests were 120, 235, and 104, respectively.

Important errors found during testing were:

(1) Failure to check for exceeding the upper limit of the total number of student records. The program did not check that the number of records entered will not exceed the set limit. When this happened, an out-of-bound reference to the student record would occur. This error was detected during the unit testing of module *EnterStudent*.

(2) Failure to reset the record counter after all the records in the database were deleted. When the user has entered some student records and subsequently deleted all these records, the program failed to reset the number of assignments entered to zero. This error was detected during system testing when trying to satisfy the SI criterion.

The test selection effort was actually quite manageable, despite the large number of tests. Over 80% of the integration tests (195) were the same as the unit tests, and many systems tests (65) were directly derived from the integration tests. It is likely that some of the duplicate tests can be avoided if we delayed the execution of the unit tests. However, this is not recommended because the earlier an error is detected, the less cost is required to fix it. Also, the set of modules that were being tested were different between unit and integration testing as integration testing involved more modules. Unit testing typically involves a few modules, drivers and stubs using tests aimed at exercising the functionality of one particular module. Integration testing generally involves more modules with tests that exercise the interactions of two modules. Thus, although some unit and integration tests were the same, they were exercising the system under slightly different conditions and were not wasting testing resources. In addition, due to the potential masking effect, the same test may detect an error during the unit testing and not during the integration testing, and vice versa.

9.1.4. Program Modifications and Regression Tests

We next describe four modifications suggested by the user. Modifications (1), (2) and (4) involved structural modification to the call graph, whereas modification (3) only required non-structural modification. Each modification was applied successively to the most recent program version. Thus, when modification (3) was made, the program version being modified had undergone both modifications (1) and (2).

Change Objective 1: Check for unique student identification numbers.

The current version of *StudentDatabase* does not check for a unique student id. The user may accidentally type in a duplicate id. It is desirable to include a check which can flag such an occurrence.

Modification 1:

Given this change objective, we came up with two possible modification alternatives. The first modification alternative was to replace the module *GetId* called by module

EnterStudent with another module which would read in a unique identification number. The second alternative was to modify the code of *EnterStudent*. The first alternative was adopted because we could limit all the code changes to one low level module and this should reduce the amount of retesting. After the change, module *EnterStudent* called module *GetUniqueid* instead of module *GetId*. Therefore, there were two modified modules.

Conceptually, there was a specification change to *EnterStudent* since the students all have unique identification numbers while some of them may have had the same identification number before the modification. However, this property was not used by other modules. Thus there was no change to *EnterStudent* from the viewpoints of the other modules. Integration testing can stop after integrating modules *GetUniqueId* and *EnterStudent*, and modules *EnterStudent* and *StudentDatabase*, respectively.

Change Objective 2: Add an option for plotting a mark distribution graph.

A desirable feature requested by the user was to be able to view the mark distribution graphically. It was decided to add an extra option to the input menu so that the user can ask for the plotting of a graph showing either the mark distribution of a particular assignment or the overall average of the students.

Modification 2:

There were four affected modules (two new and two modified modules). Modules *PlotMarks* and *PlotGraph* were introduced to perform the plotting of the mark distribution graph. Two instructions in module *Menu* were affected. Module *StudentDatabase* had two new declarations and one new instruction. Altogether, there were three new module interactions, one new software feature, two extra simple call paths, and five extra simple input paths.

Errors found during regression testing included:

(1) The new module *PlotMarks* did not check for the case when the database did not have any student record entered. This error was detected during the unit testing of *PlotMarks*.

(2) Module *PlotMarks* called module *GetAssignNo* to obtain an assignment number for plotting the mark distribution. It should instead make sure that the assignment numbers entered were below *NumberOfAssign*. A new section of code was added to check for this condition. This error was detected during the integration of modules *PlotMarks* and *GetAssignNo*.

Change Objective 3: Allow different maximum marks for each assignment.

The current version of *StudentDatabase* assumed that the maximum mark would be the same for all assignments and validation of input marks was not performed. The user requested that the program be changed such that the maximum marks of the assignments be entered by the user and each input mark be validated.

Modification 3:

This change objective required many minor modifications to eight modules: Module *StudentDatabase* had two extra declarations and two new instructions. Modules *WriteData* and *ReadData* each had two extra instructions. Module *GetAssignNo* was enlarged by five new instructions. One instruction of module *Average* was modified. Two and four instructions were added to modules *RemoveStudent* and *DisplayClass*, respectively. Finally, one instruction was modified in module *PlotMarks*. No new module was introduced. Although no new software feature was added, all the existing features were affected by the changes, and there were 16 affected module interactions.

Because of the distributive nature of this change, we expected many more tests would be required for regression testing of this modification than the previous two modifications.

A major error was found during regression unit testing of module *DisplayClass*. The lowest mark was not computed properly. This error arose because a required change was not made. The lowest mark for all the assignments were previously initialized to 10, the previous maximum mark. In making the change, this was overlooked and consequently, the program gave a lowest mark of 10 when it should have been larger. This was corrected by initializing the lowest mark to *MaxMarks*.

The second error that was missed by both unit and integration testing occurred in the module *Average*. When the marks of an assignment had not been entered, *MaxMarks* for the assignment would be initialized to 0. If module *DisplayStudent* was called, then a *divide by zero* error would be produced. It is difficult for the test analyst to test for this condition at the unit or integration testing level because of the limited scope of testing.

Change Objective 4: Assign grades to all students.

Another feature suggested by the user was the capability to assign a grade point to each student.

Modification 4:

It was decided to add a new option to the main menu which would invoke the grade assignment function. The changes included adding three new modules, one new software feature, one extra simple call path, and four extra simple input paths, and modifying five other modules. Also, one datafile was enlarged to store the grading scheme and the weights of the assignments.

One important error was found during the integration of modules *GradePoint* and *AssignDisplayGrade*. Since the program did not check for an empty database, it produced a meaningless message whenever the user asked for the assign grade option and there was no record in the database.

During the testing of the modifications, we applied the regression strategies for unit, integration, and system testing outlined in Chapters Four, Five and Six. Each affected module was analyzed and the types of changes were identified. If the specification of the module was changed, then progressive regression unit strategy was applied; otherwise, the corrective regression unit strategy was used.

For regression integration testing, we first identified all the affected modules and then the basis cases involved in integrating the modified modules and their descendants and ancestors. Test use levels were determined from Table 4 of Chapter Five. For basis cases that involved modified specification, new functional tests were created to test the new or changed functionalities.

Regression system testing was largely based on the affected features, the new calling orders, and new user inputs. The affected features were used to identify a set of functional tests that should be rerun. The affected simple call paths and simple input paths were used to identify the reusable, retestable and obsolete SI tests. New SI test were added until the SI criterion was satisfied. It was critical that tables storing feature-specification mapping, feature-module matrix, feature-test matrix, and module-test matrix be kept up-to-date so that only the relevant tests be selected to test the modifications.

9.1.5. Results

Table 6 summarizes the actual changes and the number of regression tests used for each modification. The total unit test represents the total number of new and old unit tests for testing all the affected modules. This number is typically less than the total number of unit tests used to test all the modules in the initial testing because only a few modules are affected in each modification. The total integration test represents the total number of integration tests required to test all the new and existing module interactions and the total system test denotes the total number of system tests. The sum of these three sets of tests represents the total number of regression tests used by the retest-all strategy. The last row of Table 6 gives the ratio of the number of regression tests according to our strategy to that of the retest-all strategy.

Modification	**1**	**2**	**3**	**4**
Number of Affected Source Lines	25	80	23	57
Number of Affected Modules	2	4	8	8
Number of Affected Module Interactions	2	3	16	10
Number of Affected Features	1	1	7	1
Number of Regression Unit Tests	15	22	50	40
Number of Regression Integration Tests	32	32	120	80
Number of Regression System Tests	46	24	130	38
Total Unit Test	27	40	67	66
Total Integration Test	246	278	275	307
Total System Test	106	130	130	158
Regression Tests/Total Tests	93/379 24%	78/448 17%	300/472 63%	158/531 30%

Table 6. Modification Characteristics and Regression Tests

One interesting observation is that the number of affected source lines does not seem to be the deciding factor in predicting the number of regression tests. The important factor seems to be the "extent" of a modification, which can be represented by the number of affected features and affected module interactions. Although modifications 2 and 4 involved more affected source lines than modification 3, modification 3 actually required the most regression tests. This occurred because modification 3 had the highest number of affected module interactions (16) and affected features (7) among the four modifications.

9.1.5.1. Test Effort Results

To compare the effectiveness of our strategy relative to the retest-all strategy, we reran all the tests in the test plan to check whether they would detect extra errors. No new error was discovered. Thus, we conclude that extra testing by rerunning all the tests in the current plan does not detect more errors than our strategy. As shown in the last row of Table 6, the size of the test subsets for our regression strategy relative to the retest-all strategy were 24%, 17%, 63% and 30% for the four modifications, respectively. Thus the average reduced subset was 34%. These results reveal that using our strategy requires significantly fewer tests than the retest-all strategy. The major reason for this saving is that the concept of a firewall is effective in reducing the amount of integration testing.

The savings observed in this experiment would most likely be generalized to other maintenance projects, although possibly not to the same degree. If further research establishes consistency with this result, then our strategy can be adapted to most maintenance projects.

9.1.5.2. Test Effectiveness Results

To test the effectiveness of our regression testing strategy in detecting errors, we seeded some errors in the changes and reran the set of regression tests. All seeded errors were placed in the affected modules to simulate possible errors made when the code was modified. Table 7 shows the results. A total of 13 logic errors were inserted in the four modifications. Two of these errors can be classified as extra function errors, one as a missing function error, and the rest as wrong function errors. Table 7 also shows the testing phases which detect the error. Under the Detected In column, a Y indicates that the error was detected at that testing phase; otherwise a N is shown. All errors except one were detected using our set of regression tests. 8 errors were detected during regression unit testing, 11 during regression integration testing, and 12 during regression system testing. There were three errors which were not detected during regression unit testing but were detected during regression integration testing.

The error which was not detected by our testing strategy was an extreme case of an *assignment blindness error* [81]. Some blindness errors are undetectable by any testing strategy. Our error was not detected because an incorrect program constant was used, but by coincidence it had the same value as the correct program constant. In the program, if constants *Max* and *MaxAssign* have the same values, then an error in the loop implementing the initialization of marks in modification 3 will not be detected. Since the program constants were assigned values at the declaration stage, our error was undetectable even if all program paths were exercised. This suggests that we should also test the program by assigning different values to its program constants.

The last seeded error was missed by both the unit and integration testing because they used a limited viewpoint of the system and did not consider other complicated interactions among the modules. This error was similar to the second error detected after Modification 3.

Our results confirm the expectation that some errors are likely to be missed at a certain testing phase and can only be detected in another testing phase. Although some errors can only be detected at the unit testing level, a few errors cannot be detected at this level due to the limited focus and incorrect assumption of the module's input and output. Most, but not all, of these undetected errors were discovered during integration testing when the test analyst considered a slightly larger "global viewpoint" of the software. Finally, some errors were not detected until system testing because they required complex interactions between modules that were not possible in the previous two phases of testing.

Modification	Seeded Errors	Integration Errors	Detected In		
			Unit	Integ.	Sys.
1	missing computation	wrong function	Y	Y	Y
	wrong condition	extra function	Y	Y	Y
2	missing computation	wrong function	Y	Y	Y
	wrong condition	wrong function	Y	Y	Y
	off by 1 iteration	extra function	N	Y	Y
	failure to initialize	wrong function	Y	Y	Y
3	wrong computation	wrong function	Y	Y	Y
	extra change	wrong function	N	Y	Y
(undetected)	wrong change	wrong function	N	N	N
	missing change	wrong function	N	Y	Y
4	wrong computation	wrong function	Y	Y	Y
	missing condition	missing function	Y	Y	Y
	missing condition	wrong function	N	N	Y

Table 7. Detected Seeded Errors

The results from Table 7 seem to suggest that the system tests are more effective in detecting the seeded errors than the other tests. A reason for this is that each system test usually includes several unit and integration tests because it requires the execution of system function that can involve many modules. In other words, executing all the system tests may have the same effect as executing the combined unit and integration tests. Thus, it is not surprising that the system tests tend to detect more errors than the other two types of tests. In general, the set of system tests may not be more effective than the combined sets of unit and integration tests.

9.1.5.3. Additional Results

An interesting discovery is that there are tests for unit testing which will never occur during the normal operation of the system; that is, these tests cannot be *sensitized* to the main module of the software system. For example, in unit testing of module *Swap*, tests with 0 assignment, 3 assignments, and the maximum number of assignments were used. But, in actual usage of the *StudentDatabase* program, *Swap* will not be called when no assignment mark is entered. Thus, the test of 0 assignment was superfluous. However, this test was required for satisfying the all-uses structural selection criterion for unit testing.

The above example does not imply that the all-uses criterion is faulty. It only shows that certain unit tests may not be effective in testing the actual behavior of the complete system. Besides, no test selection criteria guarantee that all tests are useful for detecting errors.

One unexpected result of this study is that the calling order graph provides more benefits than anticipated. The availability of the calling order graph not only helps with understanding and reasoning about the code, it can also help with code optimization. For example, in the implementation of module *RemoveStudent*, by looking at the calling order graph, we found that module *ClearScreen* was called on every call path from module *RemoveStudent* and it could be called once instead of being invoked at different points of the various paths. Thus, the calling order graph may be used for analyzing the program, and not just for testing purposes.

We also found that using the structural-interface criterion for system testing did not require a large testing effort. The extra effort required to generate tests to satisfy this criterion was found to be reasonable. About 10% of the systems tests were *SI* tests selected specifically to satisfy the SI criterion. Most of these tests were not difficult to create.

The interface subcriterion of the *SI* criterion was found to be easy to apply. It basically requires the test analyst to try out all the various options available to the user. Unfortunately for programs which have very few input variables and rely heavily on stored information, this criterion is not too useful for testing purposes.

The normalized relative effort for testing the program *StudentDatabase* were found to be approximately 0.25, 0.45, and 0.3 for unit testing, integration testing, and system testing, respectively. Integration testing required the most testing effort. A contributing factor to the high effort required for integration testing is the work required to code and set up various drivers, stubs and modules, and the effort needed for sensitizing the integration tests. On the other hand, there is only one entity to test for system testing, and test design can be based largely on the specification.

Although in practice most software organizations place less emphasis on integration testing and concentrate on extensive unit and system testing, our strategy has the advantage of detecting most errors before system testing starts. During both regression testing of the four modifications and the experimentation with the seeded errors, all but one of the detected errors were discovered during unit and integration testing. This suggests that by emphasizing integration testing, fewer errors are left to be discovered during system testing. Our approach is recommended because the earlier an error is detected, the less costly it is to fix. Thus, it is more desirable to detect an error during unit testing than at integration testing time, with similar advantage of detection during integration testing rather than waiting for the testing process to be completed.

9.2. Test Cost Study

Another study was conducted to validate the assumptions made in building the test cost model presented in Chapter 8. In particular, we want to establish whether the following assumptions on the selection cost Cs, execution cost Ce, and result checking cost Cc are realistic and practical:

$Cs(T) = s\,|T|$

$Ce(T) = e\,|T|$

$Cc(T) = c\,|T|$

where s, e and c represent the cost per test for the test selection, execution, and result checking, respectively, and |T| the number of tests. We collected data on the time required to select a test, execute a test, and check the test result from a large telecommunication company. Section 9.2.1 describes a survey which collected empirical data on Cs and Cc. Section 9.2.2 presents data to support the assumption that Ce may be treated as linearly dependent on the number of tests.

9.2.1. Test Selection and Result Checking Costs

A survey was recently conducted with test analysts in a large telecommunication company. The return rate of the survey was over 70%. A total of 20 responses was used in the following analysis. The respondents have between 1 to 10 years of testing experience. These test analysts were testing software from 7 projects of a recent software release. Typically, 3 to 4 test analysts were assigned to one project. The survey asked the respondents to provide answer based on their experience with the most recent software release. This provided a snap shot of the state-of-the-practice of the company.

The following questions on the survey were of particular interest to our study:

(1) After you have studied the necessary documents (specification, design etc.), how long did it take, on average, to create one test case? This includes the time to identify the test input, the expected output, pass/fail criterion, and enter this test into the test plan?

(2) How long did it take, on average, to verify a test result was correct? After running a test case, how much time was required to check that the system behaved correctly and the output was correct?

For question (1), all responses were within one minute of either 8, 13, or 18 minutes per test. We found that test analysts whose test suites contain fewer than 100 tests tended to require longer time to create a test. This can be explained by the fact that these tests were "larger" or testing more system behavior.

For question (2), the responses were within one minute of either 3, 8, or 13 minutes per test. There was no direct correlation between the test selection time and the test result checking time. In other words, a test analyst who required 8 minutes to create a test may require 3, 8, or 13 minutes to verify a test result.

The data suggests that the time for test selection can be approximated by one of 8, 13, or 18 minutes per test, and the checking time can be approximated by one of 3, 8, or 13 minutes per test in this development environment. This seems to indicate that the constants s and c in the linear assumptions of Cs and Cc vary with the type of projects and possibly on other factors. However, they only take on one of several possible values. In practice, we can treat these costs as linearly dependent on the number of tests, although further analysis is needed to determine the exact values to be used for s and c for a given project.

Note that these results are specific to a particular development environment. We believe that other organizations will also find that Cs and Cc only vary among a few values, but likely different from the figures identified in this study.

9.2.2. Test Execution Cost

To check the validity of the linear assumption of Ce, we make use of the data available in the test result database from the same telecommunication company. After each test lab shift, test analysts are required to enter the execution time of each test into a test result database. A random sample of seven projects from a recent release was chosen for the analysis to identify the relation of test execution effort and the number of tests.

Table 8 lists the number of tests used for testing and the average execution time per test. The projects can be grouped into two classes based on the nature of the applications. Projects A, B and C belong to one class while the other projects belong to another class of application. As shown in Table 8, the average execution times of projects from each class are very close to each other. In fact, the first class has an average of 12.4 minutes, a range of 1.2 minutes and a standard deviation of 0.67 minutes, while the second class has an average of 9.6 minutes, a range of 2.1 minutes and a standard deviation of 0.9 minutes. This data indicates that the execution constant e will vary slightly depending on the nature of the application, and similar applications tend to have almost identical value. Based on this limited sample, there is strong indication that the test execution time for each type of application in this software development environment can be treated as linearly dependent on the number of test cases for practical purposes.

Project	A	B	C	D	E	F	G
Number of Tests	2442	5966	6856	3866	5432	7724	8324
Average Execution Time per Test (minute)	12.8	12.7	11.6	10.7	10	8.6	9.3

Table 8. Average Execution Time per Test

Chapter Ten

Concluding Remarks

Although regression testing is an important topic, a fundamental study of the issues involved is long overdue. This dissertation studies the problem of regression testing and a general framework for regression testing is proposed. In this framework, we provide systematic procedures for selecting tests for all three phases of testing and for determining when regression testing can be terminated. We summarize the key results in the next section. Future research is described in Section 10.2.

10.1. Summary

We have identified two types of regression testing: corrective regression testing and progressive regression testing. The key difference between the two is that the specification stays the same in corrective regression testing, whereas progressive regression testing involves a modified specification. These two types involve different degrees of retesting effort. We have argued that corrective regression testing, in general, should be an easier process than progressive regression testing because more tests can be reused.

One way to reduce the retesting effort is to reuse tests in the current test plan. This entails the identification of various test classes. For regression testing purposes, tests in a test set can be grouped into five classes: reusable, retestable, obsolete, new-structural, and new-specification tests. We have identified the test classes associated with the two types of regression testing. Corrective regression testing may involve reusable, retestable, obsolete, and new-structural test classes, while progressive regression testing may involve all five test classes.

For the class of strategies that we have analyzed, we found that every regression testing strategy can be characterized by its original selection, reuse selection, and new selection criteria. The original selection criterion is used to select tests for the previous testing, the reuse selection criterion is used to choose tests from the current test set, and the new selection criterion is used to select new tests. In our framework, the reuse selection criterion is based on the all-essential instruction and all-essential module assumptions. This criterion is simple to apply and the empirical results given in Chapter Nine show that it is effective for testing purposes and does not involve many tests.

Regression testing strategies for all three phases of testing were developed and outlined in Chapters Four, Five, and Six. For regression unit testing, a framework has been laid out which includes both functional and structural testing.

Integration testing is an area that has not been thoroughly explored. Although integration strategies are well understood, the test selection problem has not been carefully studied. We have identified the common errors in software integration which usually arise because of a misunderstanding of the specification of the called module. To study this test selection problem, we have proposed a model of integration testing, with some general guidelines for selecting integration tests. Our regression integration testing strategy is based on testing a set of basis cases that model two interacting modules under a variety of change conditions. Eleven basis cases have been identified and the use levels of previous tests have been determined.

The concept of a firewall is introduced to encapsulate all the modules participating in the integration testing. The firewall imposes a limit on those modules which must be considered during regression integration testing. It identifies the area with the highest potential of error and allows the test analyst to concentrate the testing effort on those modules of the program where the error will likely be discovered. The firewall also dictates a rational sequencing of integration testing and gives a strategy on scheduling the integration order. In general, re-integration of all the modules is not required for most modifications. In Chapter Nine, we show an application of the integration basis cases and the concept of firewall to a practical program. Our experimental results indicated that this approach is valuable in reducing the number of tests required to revalidate the modified program.

For system testing, we introduce the structural-interface criterion which emphasizes testing the user input combinations and extends the notion of structural coverage in unit testing to the system level based on the calling order graph. This criterion can be viewed as both "user-oriented" and "implementation-oriented", and can be used to measure the progress of testing.

Although the testing of global variables are important to the reliability of a software system, they have been largely ignored by the testing community. We have shown that global variables can be treated like parameters for testing purposes and can be tested accordingly. Once the effect of the global variables is known, then their presence in a program does not introduce any new testing problems, other than those similar to testing parameters. However, if their effect is unknown, testing them is necessarily *ad hoc* and attaining a high degree of reliability is impossible. Strategies for testing and regression testing global variables were developed. These strategies rely heavily on the strategy for testing parameters. If the parameter testing strategy is effective, then the global variable testing strategy will also be effective.

This dissertation also presents the design of the retesting system ReTestS. ReTestS provides the following capabilities: identification of initialization requirements, test selection, test execution, test analysis and update, and test design assistance. Although developing ReTestS will require considerable effort, it is envisaged that some components of ReTestS can be developed by modifying some existing testing tools.

We also provide an answer to the question of the relative cost benefits of the selective retest and retest-all strategies. A cost model for comparing regression testing strategies has been proposed. This model is based on realistic assumptions that are supported by empirical data. We have found that the relative cost depends on four factors: the selection, the execution, the checking cost, and the ratio of the size of the selected previous test to the size of the total previous test. The conditions under which a selective retest strategy is more economical than the retest-all strategy have been established.

Since studies of large systems tend to be prohibitively expensive, we have carried out an empirical experiment on a 550-line program. Chapter Nine reports on the case study in which our concepts and strategies are applied to test this program. The results indicate that our regression testing strategy is as effective as the retest-all strategy and can save over 65 percent of the tests used by the retest-all strategy. The data reported here shows the potential savings that our strategy offers when implemented as a regression testing tool. Empirical data is also collected to support the linear assumptions made in the test cost model. Additional data is need to confirm our findings which suggests that the test selection, execution and result checking costs are linearly dependent on the number of tests.

The major contributions of this work are:

(1) A framework for making regression testing a more systematic process has been developed, and regression testing strategies for all three phases of the testing have been identified.

(2) We have identified the common errors and faults in combining modules into a working unit. Test selection strategies for integration testing and regression integration testing have been developed.

(3) The concept of the firewall is introduced and it is shown that re-integration of all the modules may not be needed for all modifications. The firewall can assist the test analyst in focusing on that part of the system where new errors may have been introduced by a correction or a design change.

(4) Testing global variables is shown to be not more difficult than testing parameters if we insist that the use of global variables be included in the specification.

(5) A regression testing system is designed which, when implemented, will reduce the current testing efforts, minimize human errors, and produce consistent-quality software.

(6) A cost model has been developed which can be used to select the most cost-effective regression testing strategy for a given change situation.

(7) Our regression testing strategy has been demonstrated on an interactive program. Our approach was able to discover all errors found by the retest-all strategy, while executing only 34% of the total number of tests.

10.2. Future Research

Several major problems remain to be addressed in regression testing. A regression testing strategy for changes in data structure has not been developed. For example, a record structure may be modified to include extra data fields. This will affect all references to the variables of this data structure. This dissertation has concentrated on functional modification and ignored the effect of data modification. Some data modifications may have serious consequences on the program and may require extensive retesting.

More research should be done on developing other reuse selection criteria. Although our reuse selection criterion, which is based on the all-essential instruction and all-essential module assumptions, provides large savings over the retest-all strategy, there may exist other reuse selection criteria which can further reduce the number of tests.

In the development of our strategy for testing global variables, we have assumed the existence of a reliable test selection method for testing parameters. Unfortunately, the parameter testing problem is an open research problem. In Chapter Seven, we have shown that this problem can be reduced to a set of four basis cases. More studies are needed to develop effective testing strategies for each of these cases.

A regression testing environment such as ReTestS should be implemented. This effort will likely be a long term project requiring approximately two man-years to complete. One possible strategy to develop ReTestS is to integrate several existing unit testing tools by providing a consistent interface and standardizing information sharing. Even by extensive use of available testing tools, many new developments are required for regression testing at the integration and system levels because minimal research has been done in these areas.

Once ReTestS is available, some large realistic software systems should be used to assess our approach in a quantitative way. We also need more experimental work to verify the assumptions, and to improve our understanding of the benefits and limitations of our strategy. In particular, the three types of integration errors identified in Chapter Five should be evaluated to see how often each occurs in practice. The concepts of integration testing basis cases and of a firewall should also be evaluated with different types of programs and under different types of maintenance scenarios.

A possible approach to regression testing is the use of dynamic assertions. These assertions are logical expressions regarding program variables and are entered into the program as comments. A preprocessor then generates and inserts the code for dynamically checking the validity of these assertions. Several papers describe ways of using dynamic assertions for testing software [18, 19, 53, 68]. It may be possible to extend these approaches to regression testing.

Another area which warrants study is the concept of a regression testing metric which can be used to estimate the retesting effort before the actual modification is made. The availability of an estimator of the retesting effort provides several important benefits. It can be used to estimate the resources, time, and effort for retesting a modified program. This is extremely valuable for scheduling the regression testing process. Also, if the maintainer comes up with several modification alternatives for a change objective, he can use the regression testing metrics to perform an analysis to find the most cost-effective alternative. All too often, the maintainer applies the first obvious modification that would achieve the change objective and does not consider the *testability* of such a change. It is possible that an alternative modification may reduce the cost of regression testing.

References

1. V. R. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation," *Communications of the ACM*, vol. 27 (1), pp. 42-52, Jan. 1984.

2. B. Beizer, in *Software system testing and quality assurance*, Van Nostrand, New York, 1984.

3. P. Benedusi, A. Cimitile, and U. De Carlini, "Post-maintenance testing based on path change analysis," *Proc. Conf. Software Maintenance*, pp. 352-361, Phoenix, 1988.

4. R. S. Boyer, B. E. Elspas, and K. N. Levitt, "SELECT - a formal system on testing and debugging programs by symbolic execution," *Proc. 1975 Conf. on Reliable Software*, pp. 234-245, Los Angeles, CA, April 1975.

5. T. A. Budd, "Mutation analysis: ideas, examples, problems and prospects," in *Computer Program Testing*, ed. B. Chandrasekaran, S. Radicchi, pp. 129-148, North-Holland, Amsterdam, 1981.

6. R. Carey and M. Bendick, "The control of a software test process," *Proc. COMPSAC 77*, pp. 327-333, Chicago, IL, Nov. 1977.

7. L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A comparison of data flow path selection criteria," *Proc. 8th Int. Conf. Software Eng*, pp. 244-251, London, UK, 1985.

8. L. A. Clarke and D. J. Richardson, "Symbolic evaluation methods - Implementations and applications," in *Computer Program Testing*, ed. B. Chandrasekaran, S. Radicchi, pp. 65-102, North-Holland, Amsterdam, 1981.

9. J. S. Collofello and J. J. Buck, "Software quality assurance for maintenance," *IEEE Software*, pp. 46-51, Sept. 1987.

10. R. A. DeMillo, R. J. Lipton, and F. G. Sawyer, "Hints on test data selection: help for the practicing programmer," *Computer*, vol. 11, pp. 34-41, April 1978.

11. F. DeRemer and H. H. Kron, "Programming-in-the-large versus programming-in-the-small," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 80-86, June 1976.

12. T. Dogsa and I. Rozman, "CAMOTE-computer aided module testing and design environment," *Proc. Conf. Software Maintenance*, pp. 404-408, Phoenix, 1988.

13. J. W. Duran and S. Ntafos, "An evaluation of random testing," *IEEE Trans. Software Eng.*, vol. SE-10 (4), pp. 438-444, 1984.

14. W. R. Elmendorf, "Cause-effect graphs in functional testing," *TR-00.2487*, IBM Systems Development Division, Poughkeepsie, N. Y., 1973.

15. R. G. Falkner, "Get the bugs out," *Service News*, pp. 39-42, Nov. 1989.

16. K. F. Fischer, F. Raji, and A. Chruscicki, "A methodology for retesting modified software," *Proc. National Telecommunications Conf.*, pp. B-6-3(1-6), New Orleans, LA, Nov. 1981.

17. P. G. Frankl, S. N. Weiss, and E. J. Weyuker, "ASSET: A system to select and evaluate tests," *Proc. of the IEEE Conf. on Software Tools*, pp. 72-79, New York, NY, April 1985.

18. J. Gannon, P. McMullin, and R. Hamlet, "Data-abstraction implementation, specification, and testing," *ACM Trans. Program Lang. Syst.*, vol. 3, pp. 211-223, 1981.

19. M. R. Garey, D. S. Johnson, D. M. Andrews, and J. P. Benson, "An automated program testing methodology and its implementation," *Proc. of the 5th International Conference on Software Engineering*, W. H. Freeman and Company, San Diego, CA, March, 1981.

20. R. Glass, in *Software Maintenance Guidebook*, Prentice Hall, Englewood Cliffs, N.J., 1979.

21. R. L. Glass, "Persistent software errors," *IEEE Trans. Software Eng.*, vol SE-7 (2), pp. 162-168, March 1981.

22. D. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, pp. 206-215, Banff, Canada, July 1988.

23. M. J. Harrold and M. L. Soffa, "An incremental approach to unit testing during maintenance," *Proc. Conf. Software Maintenance*, pp. 362-367, Phoenix, 1988.

24. M. J. Harrold and M. L. Soffa, "An incremental data flow testing tool," *6th Int. Conf. Testing Computer Software*, Washington, D.C., 1989.

25. M. J. Harrold and M. L. Soffa, "Interprocedural data flow testing," *Proc. third Symposium on Software Testing, Analysis and Verification*, pp. 158-167, Key West, 1989.

26. M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *Proc. Conf. Software Maintenance*, pp. 302-310, San Diego, 1990.

27. J. Hartmann and D. J. Robson, "Techniques for selective revalidation," *IEEE Software*, pp. 31-38, January, 1990.

28. M. S. Hecht, in *Flow analysis of computer programs*, North-Holland, Amsterdam, 1977.

29. M. S. Hecht, "A simple algorithm for global data flow analysis problems," *SIAM J. Computing*, vol. 4, no. 4, pp. 519-532, Dec. 1975.

30. E. C. van Horn, "Software must evolve," in *Software engineering*, ed. P. M. Lewis, pp. 209-226, Academic Press, 1980.

31. W. E. Howden, in *Functional program testing and analysis*, McGraw-Hill, 1987.

32. W. E. Howden, "Applicability of software validation techniques to scientific programs," *ACM Trans. Program Lang. Syst.*, vol. 2 (3), pp. 357-370, July 1980.

33. W. E. Howden, "An evaluation of the effectiveness of symbolic testing and of testing on actual data," *Software-Practice and Experience*, vol. 8, 1978.

34. W. E. Howden, "A functional approach to program testing and analysis," *IEEE Trans. Software Eng.*, vol. SE-12 (10), pp. 997-1005, Oct. 1986.

35. W. E. Howden, "Functional testing and design abstractions," *The Journal of Systems and Software*, pp. 307-313, 1980.

36. W. E. Howden, "A survey of dynamic analysis methods," *Tutorial: Software Testing and Validation Techniques*, 1981.

37. W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Software Eng.*, vol. SE-8 (2), pp. 371-379, July 1982.

38. D. C. Ince, "The automatic generation of test data," *The Computer Journal*, vol. 30, no. 1, pp. 63-69, 1987.

39. K. W. Kennedy, "A comparison of two algorithms for global data flow analysis," *SIAM J. Computing*, vol. 5, no. 1, pp. 158-180, March 1976.

40. B. Korel, "Automated software test data generation," *IEEE Trans. Software Eng.*, vol. SE-16 (8), pp. 870-879, Aug. 1990.

41. B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, pp. 155-163, Oct. 1988.

42. B. Korel and J. Laski, "A tool for data flow oriented program testing," *Proc. SoftFair II*, pp. 34-37, San Franciso, CA, 1985.

43. D. R. Kuhn, "A source code analyzer for maintenance," *Proc. Software Maintenance Workshop*, pp. 176-180, Austin, 1987.

44. J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Trans. Software Eng.*, vol. SE-9(3), pp. 347-354, 1983.

45. J. W. Laski, "On data flow guided program testing," *SIGPLAN Notices*, vol. 17, no. 9, pp. 62-71, Sept. 1982.

46. M. M. Lehman, "Programs, life cycles and laws of software evolution," *Proc. of IEEE*, vol. 68, no. 9, pp. 1060-1076, Sept. 1980.

47. H. K. N. Leung and L. White, "Insights into regression testing," *Proc. Conf. Software Maintenance*, pp.60-69, Miami, FL, Oct. 1989.

48. H. K. N. Leung and L. White, "A study of integration testing and software regression at the integration level," *Proc. Conf. Software Maintenance*, pp.290-301, San Diego, Nov. 26-29, 1990.

49. H. K. N. Leung and L. White, "A study of regression testing," *Technical Report TR-88-15,* Dept. of Computing Science, Univ. of Alberta, Canada, Sept. 1988.

50. B. P. Lientz and E. B. Swanson, in *Software Maintenance Management*, Addison-Wesley, 1980.

51. B. P. Lientz and E. B. Swanson, "Characteristics of application software maintenance," *Comm. ACM*, vol. 21, no. 6, pp. 466-471, 1978.

52. Guideline on Software Maintenance, in *Federal Information Processing Standards*, U.S. Dept. Commerce/National Bureau of Standards, Standard FIPS PUB 106, June 1984.

53. P. R. McMullin, J. D. Gannon, and M. D. Weiser, "Implementing a compiler-based test tool," *Software-Practice and Experience*, vol. 12, pp. 971-979, 1982.

54. L. J. Morell, "Theoretical insights into fault-based testing," *Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, pp. 45-62, Banff, Canada, July 1988.

55. S. S. Muchnick and N. D. Jones, in *Program flow analysis: Theory and Applications*, Prentice-Hall International, 1981.

56. G. J. Myers, in *Software reliability: principles and practices*, Wiley-Interscience, New York, 1976.

57. G. J. Myers, in *The art of software testing*, Wiley-Interscience, New York, 1979.

58. K. Narayanaswamy and W. Scacchi, "Maintaining configurations of evolving software systems," *IEEE Trans. Software Eng.*, vol. SE-13, no. 3, pp. 324-334, March 1987.

59. S. Ntafos, "On required element testing," *IEEE Trans. Software Eng.*, vol. SE-10 (6), pp. 795-803, 1984.

60. B. Raither and L. Osterweil, "TRICS: a testing tool for C," *Proc. First European Software Engineering Conf.*, pp. 254-262, Strasbourg, France, Sept. 1987.

61. S. Rapps and E. J. Weyuker, "Data flow analysis techniques for program test data selection," *Proc. Sixth International Conference on Software Eng.*, pp. 272-278, Tokyo, Japan, Sept. 1982.

62. S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Software Eng.*, vol. SE-11 (4), pp. 367-375, 1985.

63. D. J. Richardson, "A partition analysis method to demonstrate program reliability," *Ph.D dissertation, Univ. Masachusetts, Amherst,* Sept. 1981.

64. D. J. Richardson and L. A. Clarke, "Partition analysis: a method of combining testing and verification," *IEEE Trans. Software Eng.*, vol. SE-11 (12), pp.1477-1490, 1985.

65. D. J. Richardson and M. C. Thompson, "The RELAY model of error detection and its application," *Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, pp. 223-230, Banff, Canada, July 1988.

66. *Software Engineering Automated Tools Index*, Software Research Associates, San Francisco, 1982.

67. W. P. Stevens, G. Myers, and L. L. Constantine, "Structured Design," *IBM Systems Journal,* vol. 13, 1974.

68. L. G. Stucki, "New directions in automated tools for improving software quality," in *Current trends in programming methodology (Vol. II)*, ed. R. Yeh, Prentice-Hall, Englewood Cliffs, N. J., 1977.

69. H. G. Stuebing, "A modern facility for software production and maintenance," *Proc. COMPSAC 80*, pp. 407-418, Chicago, IL, 1980.

70. H. G. Stuebing, "A software engineering environment (SEE) for weapon system software," *IEEE Trans. Software Eng.*, vol. SE-10 (4), pp. 384-397, July 1984.

71. D. Talbot and R. W. Witty, *Alvey programme software engineering strategy,* Alvey Directorate, London, 1983.

72. J. D. Ullman, "A survey of data flow analysis techniques," *Proc. 2nd USA-Japan Comp. Conf.*, Tokyo, Japan, Aug. 1975.

73. E. J. Weyuker, "Axiomatizing software test data adequacy," *IEEE Trans. Software Eng.*, vol. SE-12 (12), pp. 1128-1138, Dec. 1986.

74. E. J. Weyuker, "The complexity of data flow criteria for test data selection," *Inf. Process. Lett.*, vol. 19 (2), pp. 103-109, 1984.

75. E. J. Weyuker, "An empirical study of the complexity of data flow testing," *Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, pp. 188-195, Banff, Canada, July 1988.

76. E. J. Weyuker and T. J. Ostrand, "Theories of program testing and the application of revealing subdomains," *IEEE Trans. Software Eng.*, vol. SE-6 (3), pp. 236-246, 1980.

77. L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Software Eng.*, vol. SE-6 (3), pp. 247-257, 1980.

78. S. S. Yau and J. S. Collofello, "Design stability measures for software maintenance," *IEEE Trans. Software Eng.*, vol. SE-11 (9), pp. 849-856, Sept. 1985.

79. S. S. Yau and Z. Kishimoto, "A method for revalidating modified programs in the maintenance phase," *Proc. COMPSAC 87*, pp. 272-277, Tokyo, Japan, 1987.

80. K. Yue, "What does it mean to say that a specification is complete," *Proc. 4th Int. Workshop on Software Specification and Design*, pp. 42-49, Monterey, CA, April, 1987.

81. S. J. Zeil, "Testing for perturbations of program statements," *IEEE Trans. Software Eng.*, vol. SE-9 (3), pp. 335-346, May 1983.

82. M. V. Zelkowitz, "Perspective on software engineering," *ACM Computing Surveys*, vol. 10, no. 2, pp. 197-216, June 1978.

Appendix I

An Overview of Integration Strategies

Among the many integration strategies, some assume each module is unit tested before the integration and others combine unit testing and integration into a single step. There are eight common integration strategies. The first five strategies presented below are *incremental* strategies. They merge a set of modules (usually just one module) at a time to the set of previously tested modules. The set of merged modules increases incrementally. The sixth strategy is a *nonincremental* approach (or a *phased* approach) because all the modules are grouped together simultaneously and tested. The seventh strategy is a mixed strategy which includes both the features of incremental and nonincremental strategies. The last one can be viewed as a special version of the top-down strategy. These strategies can best be illustrated with an example. We will use a section of the call graph of program *StudentDatabase*, shown in Figure I.1, as the running example.

Bottom-up Testing

In this approach the program is merged and tested from the bottom to the top of the call graph. All the *terminal modules* are unit tested in isolation, where a *terminal module* is a module which does not call another module. The next higher level modules are then tested, one at a time, with these tested modules. The next module to be tested must have all its called modules in the set of previously tested modules. This process is repeated until the top module is merged. In our example, modules CS, S, A and GAN are each tested as a stand-alone entity. Module SON is then added to the set of tested module and the resulting set is then tested. This incremental process continues until the top module (SD) is added and tested. If integration testing is done sequentially, one possible order of integration is shown in Table I.1. Driver(M) represents the driver module of module M and stub(M) denotes the stub of module M.

A *testing environment* for testing a module interaction between modules A and B is defined to be A, B, and all the modules, drivers, and stubs which participate in the testing. The same testing environment may be used to test several module interactions. Different integration strategies will likely have different testing environments for testing the same module interaction.

Top-down Testing

This strategy starts with the top module in the call graph and adds one module at a time to the set of merged modules. The top module is the only one which is unit tested in isolation. There are two possible strategies in picking the next module to be merged. The first strategy requires that the next module must have all its calling modules integrated. The second strategy relaxes the previous requirement by picking any module which has at least one of its calling modules tested previously. We will call the first strategy *strict top-down* and the second *non-strict top-down testing*. Table I.2 shows a possible strict top-down integration order for our example.

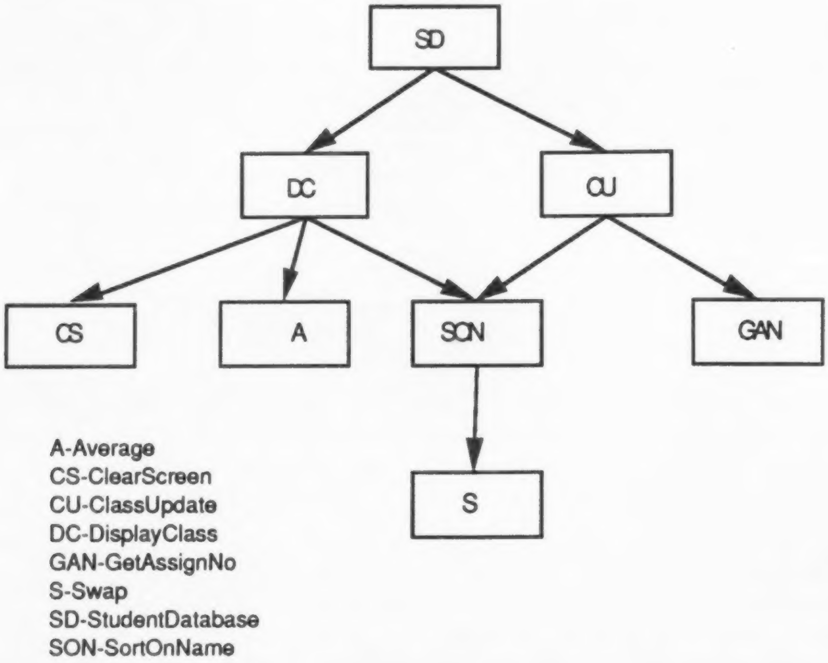

SD
DC
CU
CS
A
SON
GAN
S
A-Average
CS-ClearScreen
CU-ClassUpdate
DC-DisplayClass
GAN-GetAssignNo
S-Swap
SD-StudentDatabase
SON-SortOnName

Figure I.1. A Section of the Call Graph of Program StudentDatabase

Order	Testing Environment	Module Interaction
1	CS, driver(CS)	Unit test CS
2	A, driver(A)	Unit test A
3	S, driver(S)	Unit test S
4	GAN, driver(GAN)	Unit test GAN
5	S, SON, driver(SON)	S-SON
6	CS, A, SON, S, DC, driver(DC), driver(SON)	CS-DC, A-DC, SON-DC
7	S, SON, GAN, CU, driver(CU), driver(SON)	SON-CU, GAN-CU
8	CS, A, S, GAN, SON, DC, CU, SD	DC-SD, CU-SD

Table I.1. Integration Order for Bottom-up Testing

Order	Testing Environment	Module Interaction
1	SD, stub(DC), stub(CU)	Unit test SD
2	SD, DC, stub(CU), stub(CS), stub(A), stub(SON)	SD-DC
3	SD, DC, CU, stub(CS), stub(A), stub(SON), stub(GAN)	SD-CU
4	SD, DC, CU, CS, stub(A), stub(SON), stub(GAN)	DC-CS
5	SD, DC, CU, CS, A, stub(SON), stub(GAN)	DC-A
6	SD, DC, CU, CS, A, SON, stub(GAN), stub(S)	DC-SON, CU-SON
7	SD, DC, CU, CS, A, SON, GAN, stub(S)	CU-GAN
8	SD, DC, CU, CS, A, SON, GAN, S	SON-S

Table I.2. Integration Order for Top-down Testing

Modified Top-down Testing

One problem with top-down testing is that it may be impossible to test certain logical conditions within the merged program such as error checks conducted in individual modules. This may prevent thorough testing of particular modules. The modified top-down testing is designed to overcome this problem. This strategy is similar to the top-down testing strategy except that it requires every module to be unit tested in isolation before it is integrated into the program. As in the case of top-down testing, we have strict and non-strict modified top-down testing, depending on the conditions for selecting the next module to be merged.

Sandwich Testing

This strategy combines both the top-down and bottom-up testing. The top-down and bottom-up testing are applied simultaneously and the program is integrated from both the top and the bottom of the call graph. Eventually the integration meet somewhere in the middle of the call graph. Table I.3 gives a possible integration order using the sandwich testing strategy in our example.

Modified Sandwich Testing

As the name implies, this strategy is a modified version of sandwich testing. All modules are unit tested before applying the sandwich testing strategy. This extra step overcomes the problem described earlier with top-down testing.

Big-bang Testing

In this approach, each module is first unit tested in isolation. All the modules are then merged together at once and tested; thus the name *big-bang*.

Order	Testing Environment	Module Interaction
1	SD, stub(DC), stub(CU)	Unit test SD
2	SD, DC, stub(CU), stub(CS), stub(A), stub(SON)	SD-DC
3	SD, DC, CU, stub(CS), stub(A), stub(SON), stub(GAN)	SD-CU
4	CS, driver(CS)	Unit test CS
5	A, driver(A)	Unit test A
6	S, driver(S)	Unit test S
7	GAN, driver(GAN)	Unit test GAN
8	S, SON, driver(SON)	S-SON
9	SD, DC, CU, CS, A, SON, GAN, S	DC-CS, DC-A, DC-SON, CU-SON, CU-GAN

Table I.3. Integration Order for Sandwich Testing

Mixed-bag Testing

The overall strategy is bottom-up, but top-down is also used when the control structures are complicated. Finally big-bang strategy is used to test the *backbone,* which is a partially integrated system that contains the facilities to perform input and output, memory resources allocation and de-allocation, event recording, and run controls. Integration is done in blocks that encompass several levels of the call graph, rather than one level at a time.

Build Testing

Build testing integrates the modules according to *builds*. A *build* is a set of functionally related modules of the software system. Each module in a build is unit tested and integrated. The builds are then integrated one-by-one into the program. The order of integration is based on the technical risks of each software feature. Modules which implement critical software features are put in the first build and they will be integrated first. Less *important* modules are integrated later. A major advantage of this strategy is that the program is operational with increasing capabilities as integration of successive builds continues. This strategy can be viewed as a non-strict modified top-down testing which tests the software features in an order according to their technical risks and characteristics. As an example, assume that modules SD, DC, SON and S implement a critical software feature, modules SD, DC, CS and A implement a less critical software feature, and modules SD, CU, and GAN implement the least critical software feature. Modules SD, DC, SON and S will be put in the first build and will be integrated first. The second build will include modules CS and A. These modules will be integrated next. The last build contains modules CU and GAN, which will be the last ones to be integrated with the rest of the modules. Table I.4 shows the integration order.

Observe that there are only three major strategies among the eight common strategies: top-down, bottom-up, and big-bang. The other strategies are simply compromises and variations of these three. A major disadvantage of non-incremental testing is that it is hard to pinpoint the module which may contain the fault when a test fails. Thus, we assume these strategies (that is, big-bang, mixed bag) will not be used in a systematic testing process.

Order	Testing Environment	Module Interaction
1	SD, stub(DC), stub(CU)	Unit test SD
2	SD, DC, stub(CS), stub(A), stub(SON), stub(CU)	SD-DC
3	SD, DC, SON, stub(CS), stub(A), stub(S), stub(CU)	DC-SON
4	SD, DC, SON, S, stub(CS), stub(A), stub(CU)	SON-S
5	SD, DC, SON, S, CS, stub(A), stub(CU)	DC-CS
6	SD, DC, SON, S, CS, A, stub(CU)	DC-A
7	SD, DC, SON, S, CS, A, CU, stub(GAN)	SD-CU, CU-SON
8	SD, DC, SON, S, CS, A, CU, GAN	CU-GAN

Table I.4. Integration Order for Build Testing

Appendix II

Independence of Regression Testing and Integration Strategies

This appendix shows that our regression testing strategy is independent of the incremental integration strategy used in the previous testing cycle. In the original integration testing, we assume that there are sufficient tests for testing the module interactions. If we can duplicate the same *integration testing environment* for regression testing, and the software passes all the tests selected by the test selection criteria, then we achieve the same "degree of testing" as before the modification.

The key in showing that the regression testing strategy is independent of the incremental integration strategy is to recognize that the regression strategy is directly dependent on the testing environment and not on a particular integration strategy. If the testing environment for testing each module interaction is stored during testing, then during regression testing it is immaterial to know the integration strategy used earlier because the modified module interaction can be tested by retrieving its testing environment. The testing environment encapsulates all the information that we need to know about the previous integration strategy.

During regression integration testing, any stub of a module can be replaced by that module and all its descendants since these modules have been integrated in the previous testing, and the stub is supposed to emulate their behavior (at least for the applied tests).

There are two cases when there are no previous testing environments: adding a new module and deleting an existing module. We next analyze these cases:

(1) Adding a new module

For simplicity of presentation, we will assume that the new module B is called by only one module (A) and calls only one other module (C). There are two module interactions to be tested: A and B, and B and C. There are two possible orderings of integration: test module interaction between A and B, then that of B and C, or vice versa. Any integration strategy will use one of these two orderings. The order does not affect the effectiveness of the testing process. We describe an integration strategy below.

When testing the module interaction of A and B, we need a driver for A and a stub for C. For the stub of C, we can just use C and its descendants since they are tested and are available. For the driver for A, we have an option of using all the ancestors of A or creating a new driver. The former approach has the advantage that no new code needs to be created but it may be difficult to create tests to traverse A from the top module of the call graph. A new driver will simplify testing since it is easier to design test input to the driver. We will use the former approach because it is the same as top-down integration. Thus the testing environment for the module interaction of A and B includes all the modules of the system.

When testing the module interaction of B and C, we need a driver for B and stubs for the descendants of C. But since module A and its ancestors are available, they can be used as a driver. Also, all the descendants of C are available and are tested, so we can just use them rather than creating new stubs. Integration using the descendants of C can be viewed as bottom-up testing. The testing environment includes all the modules of the system.

For the general case when B calls several modules and is also called by other modules, similar arguments as above can be applied. The testing environment again includes all the modules of the system. The major difference is that more module interactions need to be tested.

(2) Deleting a module

Let the deleted module be denoted by E. There are two basic cases to be considered:

(2a) A module interaction is deleted and no new module interaction is introduced.

(2b) A module interaction is deleted and a new module interaction involving the immediate ancestor of E is created.

In case (2a), no integration testing is needed, but all testing environments which include E must be updated by removing E. Case (2b) is similar to the last step of the Sandwich integration strategy when the top and bottom portions of the call graph are to be integrated together. The testing environment includes all the modules of the system. All testing environments which include E should also be updated by removing E.

For both cases (1) and (2), we can use all the modules as a testing environment for the affected module interactions and this regression testing strategy obviously does not depend on the previous integration strategy.

In the analysis of Chapter Five, we have assumed that all modules are unit tested before the beginning of the integration process. This is a common and effective practice since it is realistic to have each code developer to unit test his own modules before trying to integrate different modules together. But in some strategies, such as top-down integration, unit testing of certain modules is not performed. We next consider modifications to our strategy for these situations.

Let module A be the calling module which calls module B. Consider the case of bottom-up integration when the non-terminal modules are not unit tested. We create f_B when testing module B and its descendants. Now, in testing the module interaction between A and B, we first create f_A which includes all $f_{A.B}$. We then try to sensitize f_B to module A, generating $f_{B<A}$. Thus integration testing includes the same tests $f_{A.B} \cup f_{B<A}$. The only difference between unit testing and no unit testing is that the testing environment for generating f_A and f_B are different. The case for top-down integration can be similarly shown to include $f_{A.B} \cup f_{B<A}$, with f_A and f_B generated under different testing environments from unit testing, followed by integration testing. Thus, the only change to our strategy when the modules are not unit tested is that the testing environment in generating the functional tests may be different.

Appendix III

Glossary of Terms and Symbols

Affected module: A modified, new, or deleted module.

Ancestor: Module A is an *ancestor* of module B if there exists a sequence of calls in the call graph from A to B.

Call graph: A graph showing the control hierarchy of the program with rectangles representing modules. An arrow from module A to module B indicates that A may call B.

Call instruction: An instruction which invokes another module.

Chain of calls: A sequence of calls Call(M_1), ...,Call(M_k) such that module M_i may call module M_{i+1}, $1 \le i \le k-1$.

Change information: The change information specifies the modules and/or the specification which are modified.

Code change: Either a module change or an instruction change.

Common ancestor: A *common ancestor* C of modules A and B is a module in the call graph which satisfies the following conditions: (1) C and A are on the same chain of calls and C is called before A; (2) C and B are on another chain of calls and C is called before B.

Component-test matrix: A matrix which records the program components exercised by each test. The component-test matrix is represented by $[A_{ij}]$, $1 \le i \le c$, $1 \le j \le t$, where c is the total number of components, t the total number of tests, $A_{ij} = 1$ if test j traverses component i, and $A_{ij} = 0$ otherwise.

Define-use module pair: A *define-use module pair* is a defining module A and a using module B of the same variable v such that the definition of v in A may reach B.

Defining module: A *defining module* of a variable is a module which directly defines the variable.

Definition: A *definition* of variable v is an instruction which assigns a value to v.

Definition-clear path: A path (n_x, n_i, ..., n_j, n_y) containing no definition of variable v in nodes n_i, ..., n_j is called a *definition-clear path* with respect to v from node n_x to node n_y.

Definition-use relation: A definition-use relation exists from instruction J to instruction K if there exists a variable v such that (1) $v \in$ DEF(J), (2) $v \in$ USE(K), and (3) there exists a definition-clear path from J to K w.r.t. v.

Descendant: Module B is a *descendant* of module A if there exists a sequence of calls in the call graph from A to B.

Directly defined: A global variable or a parameter is *directly defined* in a module M if it is defined by an instruction other than a call instruction in M.

Directly referenced: A global variable or a parameter is *directly referenced* in a module M if it is directly used or directly defined in M.

Directly used: A global variable or a parameter is *directly used* in a module M if it is used by an instruction other than a call instruction in M.

Error: A mental mistake by a programmer or designer.

Extremal tests: Tests which are made up with the extreme values of the input variables.

Failure: A *failure* occurs whenever the software system fails to perform its required function according to its specification.

Fault: A software defect which can cause a failure.

Feasible: A path is *feasible* if there exists input data which causes the path to be traversed during program execution.

Global variable oracle: A relation that specifies acceptable behavior of the global variable.

Incremental integration strategy: A strategy which merges a set of modules (usually just one module) at a time to the set of previously tested modules.

Independent: Instruction J is *independent* of instruction I if J is not in the scope of influence of I and I is not in the scope of influence of J.

Instruction change: Any change to an instruction J, a deletion of J, or an addition of a new J, which accomplishes a change objective.

Instruction trajectory: A feasible path that has actually been executed for some input.

Kill: A definition of variable v which reaches a module M is *killed* by M if M redefines v on all paths through M.

Least common ancestor: A *least common ancestor* L of modules A and B is a common ancestor of A and B such that no descendant of L is also a common ancestor of A and B.

Module change: Either a set of instruction changes of a module M, a deletion of M, or an addition of a new M, which accomplishes a change objective.

Module trajectory: A trace of modules executed by a single test.

Nonincremental integration strategy: A strategy which groups all the modules together simultaneously and tests them.

Non-structural modification: The modification does not affect the call graph.

Path: A subpath that begins at the start node and ends at the final node of the control flow graph.

Program change: A set of module changes which accomplishes a change objective.

Program component: Any subset of instructions of a program.

Reliability: Software *reliability* is the probability that a software system will operate without failure for a specified time in a specified environment.

Scope of influence: Instruction K is in the *scope of influence* of instruction J if (1) there is a direct or indirect definition-use relation from J to K, or (2) if J is a conditional instruction, the execution of K depends on the outcomes of J.

Sensitized: Inputs to module B are *sensitized* to its calling module A if the corresponding inputs to A which cause the required inputs to B can be generated.

Software feature: A specific function in the software system and can be identified from the software specification; it is usually implemented by a group of modules.

Structural modification: The edges and nodes of the call graph may be added, removed, or changed.

Subpath: a subpath from n_i to n_{i+k-1} of length k is a list of nodes (n_i, ..., n_{i+k-1}) in the control flow graph such that for all j, $i \le j \le i+k-1$, $(n_j, n_{j+1}) \in E$, where E represents the set of edges of the control flow graph.

Subsume: A test criterion C *subsumes* another test criterion D if any test set which satisfies C will also satisfy D.

Testing guideline: A complete specification of the testing process giving the test design strategy, the coverage measure achieved, and the procedure for handling obsolete tests.

Transferring module: A transferring module of variable v is the module M which satisfies the following conditions: (1) v is an input parameter of M. (2) M is not a using or a defining module of v.

Use: A *use* of variable v is an instruction in which this variable is referenced.

Using module: A *using module* of a variable is a module which directly uses the variable.

A->B(p_i): Module A calls module B and passes an input parameter p to B.

A->B(p_o): Module A calls module B and B returns an output parameter p to A.

Call(B): A call instruction to module B.

CodeCh(A): Module A has undergone code modification but its specification is not modified.

DEF(J): A set of variables whose values are defined in instruction J.

def(M,v): Module M directly defines the variable v.

f_A: Functional tests of module A.

$f_{A.B}$: Functional tests of module A which also traverse the call to module B.

$f_{B<A}$: Functional tests of module B which can be sensitized to its calling module A.

I(Su_j): the set of instructions for implementing Su_j.

Ia(SM): the set of instructions modified or deleted due to the modification to SM.

Ia(SD): the set of instructions modified or deleted due to the deletion of SD.

Ia(SA): Ia(SM) $\cup$ Ia(SD).

MP: the set of all program paths in a module.

NoCh(A): Module A is not modified.

n_i: A node in a control flow graph.

(n_i, n_j): An edge in a control flow graph.

NsT: New-structural tests.

NpT: New-specification tests.

OT: Obsolete tests.

RuT: Reusable tests.

RtT: Retestable tests.

SpecCh(A): Module A has undergone specification modification.

S => R: The set R of instructions is in the *scope of influence* of the set S of instructions.

S || R: The set R of instructions is *independent* of the set S of instructions.

S_i: Interpreted specification.

S_a: Actual specification.

Su: the design specification of a module, Su = {Su_1, ..., Su_k}, where k is the total number of subspecs in the module.

SM: the set of subspecs to be modified.

SD: the set of subspecs to be deleted.

SA: the affected subspecs, SA = SM $\cup$ SD.

SW: the set of new subspecs added to the module.
trans(M,v): Module M is a transferring module of the variable v.
$T_F(Su)$: the set of specification-based (functional) tests.
$T_S(Su)$: the set of structural-based tests.
$T(Su)$: $T_F(Su) \cup T_S(Su)$.
$T_F(Su_j)$: the set of specification-based tests for testing subspec Su_j.
$T_F(SM)$: the set of specification-based tests for testing SM.
$T_F(SD)$: the set of specification-based tests for testing SD.
$T_F(SA)$: $T_F(SM) \cup T_F(SD)$.
use(M,v): Module M directly uses the variable v.
USE(J): A set of variables whose values are used in instruction J.

VITA

NAME: Hareton Kam Nang Leung

PLACE OF BIRTH: Hong Kong

POST-SECONDARY EDUCATION:

M.Sc., Computing Science, May 1983, Simon Fraser University

B.Sc. (Honours), Physics and Astronomy, May 1980, University of British Columbia

AWARDS:

NSERC Postgraduate Scholarship, 1983

The Sir Charles Tupper I.O.D.E. Memorial Scholarship, 1975

PUBLICATIONS:

H.K.N. Leung and L. White, "A cost model to compare regression test strategies", *Conf. on Software Maintenance*, Oct. 1991, Sorrento.

H.K.N. Leung and L. White, "Insights into testing and regression testing global variables", *Journal of Software Maintenance*, vol. 2, no. 4, Dec. 1990, pp209-222.

H.K.N. Leung and L. White, "A study of integration testing and software regression at the integration level", *Conf. on Software Maintenance*, Nov. 1990, San Diego, pp290-301.

H.K.N. Leung and L. White, "Insights into regression testing", *Conf. on Software Maintenance*, Oct. 1989, Miami, Florida, pp60-69.

H.K.N. Leung and H. Reghbati, "Comments on program slicing", *IEEE Trans. on Software Eng.*, 13(12), Dec. 1987, pp1370-1371.

E.M. Shoemaker and H.K.N. Leung, "Subglacial drainage for an ice sheet resting upon a layered deformable bed", *J. of Geophysical Research*, Vol 92, No. B6, May 1987.